SYNOPSYS®

DesignWare®

Developers Guide

U-2022.12
December 2022

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
www.synopsys.com

Contents

Preface

1.1 About This Manual

DesignWare Developer is a tool that enables you to encapsulate your own designs as DesignWare components, also called IP. This manual explains what components are and how to create your own IP.

1.1.1 Audience

This manual is for silicon suppliers, systems companies, and third party macro and model development houses who want to develop technology-specific designs or proprietary DesignWare components. Readers should be familiar with Synopsys synthesis tools and have a basic knowledge of VHDL or Verilog.

1.1.2 Related Documents

This manual is part of the DesignWare document set installed at `$SYNOPSYS/dw/doc`. For additional information about DesignWare Building Block IP (formally called DesignWare Foundation Library), refer to:

- *DesignWare Building Block IP User Guide*

For web-based access to DesignWare Building Blocks IP, see:

- https://www.synopsys.com/dw/buildingblock.php

1.1.3 Manual Overview

This manual contains the following chapters and appendixes:

"Preface" (this chapter)	Describes the manual and lists the typographical conventions and symbols used in it; tells how to get technical assistance.
"DesignWare Concepts"	Address the following questions: "What are DesignWare Building Block IP Components?" "How are Building Block IP Components Used?" "How Do I Create Building Block IP Components?"
"Tutorial"	Takes you step-by-step through the process of creating DesignWare Building Block IP components from existing design data.
"Creating DesignWare Implementations"	Describes how to prepare your design descriptions for use as DesignWare implementations.
"Creating Synthetic Libraries: Basic"	Covers the following topics: "Anatomy of a Synthetic Library" "Writing Synthetic Library Code" "Mapping HDL Operators to Your Components"
"Creating Synthetic Libraries: Advanced"	Describes "Legality and Priority" and "Optional Pins"
"Compiling and Verifying Synthetic Libraries"	Describes "Compiling Synthetic Library Code" and "Verifying Implementations and Bindings"
"Distributing DesignWare Building Block IP"	Provides information for developers of DesignWare Building Block IP components who want to distribute the components they develop to outside customers.
"Standard Synthetic Operators"	Provides a table that lists the HDL operators that are mapped to synthetic operators in the Synopsys standard synthetic library standard.sldb.

1.2 Synopsys Common Licensing (SCL)

You can find general SCL information on the Web at:

http://www.synopsys.com/keys

1.3 Getting Help

If you have a question about using Synopsys products, please consult product documentation. You can also access documentation for DesignWare products on the Web as follows:

- SolvNetPlus provides access to all Synopsys support, documentation, training, and more:

  https://solvnetplus.synopsys.com

- To search for DesignWare IP components, downloads, documentation, and more :

  https://www.synopsys.com/designware-ip.html

- To contact a Synopsys Support Center online or by phone:

  https://www.synopsys.com/support/global-support-centers.html

1 DesignWare Concepts

It is widely recognized that design teams can increase both efficiency and quality of results by reusing designs. The DesignWare family includes the following products (among others):

- DesignWare Library - contains the principal ingredients for design and verification including high speed datapath components, AMBA® On-Chip Bus, memory portfolio, verification models of standard bus & I/O, popular Star IP processors and Board verification IP. AMBA is a registered trademark of ARM Limited and is used under license.
- DesignWare Verification Library- a subset of the DesignWare Library and contains reusable, pre-verified verification IP of the industry's most popular bus and interface standards such as AMBA, PCI Express, PCI-X, PCI, USB On-the-Go, Ethernet, I^2C and thousands of memory models.
- DesignWare Cores - silicon-proven, digital and analog standards- based connectivity IP such as PCI Express, PCI-X, PCI, USB 2.0 On-the-Go (OTG), USB 2.0 PHY, USB 1.1 and Ethernet.

DesignWare Developer, part of the DesignWare product family, lets you create DesignWare Building Block IP from your own design data. By turning your design data into DesignWare Building Block IP, you provide other designers (your end-users) with several advantages. For example, when DesignWare Building Block IP components are included in a design, they are subject to the high-level optimizations performed by the synthesis software (resource sharing, arithmetic optimization, and implementation selection).

This chapter addresses the following questions:

- "What is DesignWare Building Block IP?" on page 11
- "How are DesignWare Building Block IP Used?" on page 14
- "How Do I Create DesignWare Building Block IP Components?" on page 16

1.1 What is DesignWare Building Block IP?

DesignWare Building Block IP components are verified design units that have been provided with hooks for integration into the Synopsys synthesis environment.

The rest of this chapter expands on this basic definition.

1.2 The Parts of DesignWare Building Block IP

A DesignWare Building Block IP component consists of the following:

- *Verified, synthesis-processable design description(s)* – These design descriptions represent the intellectual property you want to capture for reuse.

  The designs must be *synthesis-processable*: capable of being read into the Synopsys synthesis environment. The style of description can vary from a technology-specific netlist or hard macro that will not be altered by synthesis, up to a full hierarchical HDL description of a parameterizable, optimizable design.

- *Modeling, compilation, and licensing directives* – One of the main advantages of DesignWare is that you can create many different design descriptions (implementations) for a given function; your end users can let the synthesis tools choose which implementation to use in any given context.

  You apply modeling directives to your design descriptions to control how Design Compiler models your designs during implementation selection.

  Compilation directives control how (and whether) Design Compiler optimizes your DesignWare Building Block IP component during the optimization of circuits that contain the component.

  Licensing directives protect your designs from unauthorized use.

- *Links that connect your design to the Synopsys synthesis environment* – When your end user includes an instance of your DesignWare Building Block IP component in a circuit, the component undergoes a chain of transformations when the circuit is synthesized. The process starts with a simple HDL construct—an operator or a component instantiation—supplied by the user. The final result depends on the nature of implementation the tools select as the best: for synthesizable HDL, the result is an optimized netlist in a particular technology; for a macro, the result is simply an instance of the macro.

- As a developer of DesignWare Building Block IP components, you set up the links required for this chain of transformations by writing a *synthetic library* description. The syntax for synthetic library descriptions is similar to that used by Library Compiler.

This book assumes that you are starting with verified designs in synthesis-processable form. It tells you how to encapsulate your designs for reuse by applying the necessary directives and by writing synthetic library code.

1.3 The DesignWare Paradigm: Some Key Terms

The DesignWare paradigm is built on a hierarchy of abstractions: HDL operators (either built-in operators like `+` and `*`, or HDL functions and procedures) are associated with synthetic operators, which are bound in turn to synthetic modules. Each synthetic module can have multiple architectural realizations, called implementations (Figure 1-1). The next few sections explain these terms.

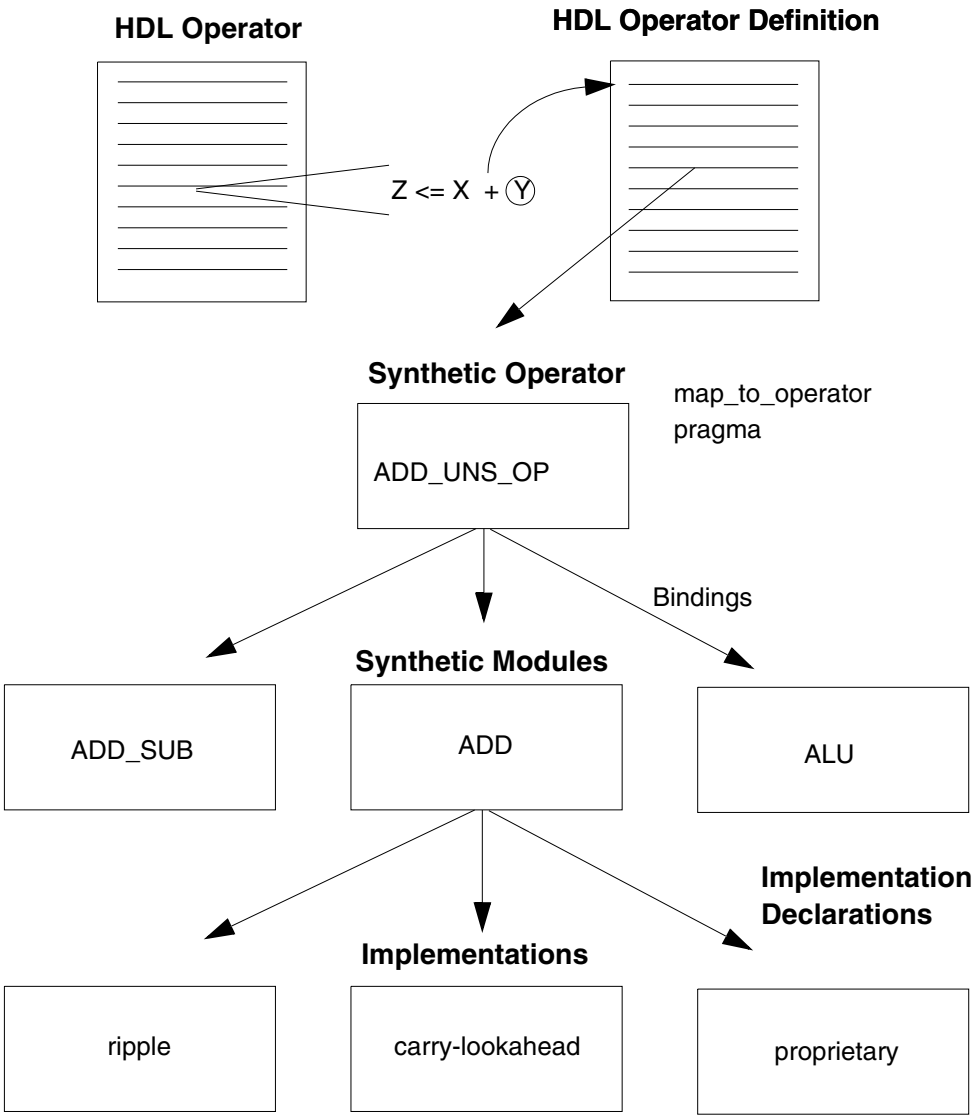
1.3.1 HDL Operators

An HDL operator is a VHDL or Verilog construct that manipulates data input values to produce output values. Some HDL operators are built into the language, like `+`, `-`, and `*`; user-defined subprograms (functions and procedures) are also considered operators.

Each operator that is not built into the language has a definition written in HDL. In VHDL, several related operator definitions are often stored together in a package. Each definition contains a simulatable

specification for the operator behavior, and, for DesignWare, a `map_to_operator` pragma that links the HDL operator to an equivalent synthetic operator.

Figure 1-1 DesignWare Hierarchy



1.3.2 Synthetic Operators

A *synthetic operator* is an abstraction of the operation called for by the HDL operator. This abstraction makes it possible for the synthesis tools to perform high-level optimizations while deferring the binding of the operation to a particular synthetic module.

The Synopsys synthesis tools can perform *arithmetic* and *resource sharing* optimizations on circuits that contain synthetic operators. Arithmetic optimization uses the rules of algebra to rearrange operations. Resource sharing allows similar operations that do not overlap to be carried out by the same physical

hardware. For more information about high-level optimization, see the *HDL Compiler for Verilog Reference Manual* or the *VHDL Compiler Reference Manual*.

1.3.3 Synthetic Modules

A *synthetic module* (not to be confused with the term "module" in Verilog) defines a common interface for a family of implementations. All implementations of a given module must have the same ports and the same input-output behavior.

Synthetic operators are associated with synthetic modules through declarations called *bindings*. For example, a binding associates the synthetic operator for addition with the adder module (you can also say that the synthetic addition operator is *bound* to the adder module).

More than one synthetic operator can be bound to a given synthetic module, and each operator can be bound to more than one module. You declare bindings in your synthetic library code.

1.3.4 Implementations

An *implementation* is a synthesis-processable design description that is linked to a synthetic module in the DesignWare hierarchy. You declare implementations and associate them with modules in your synthetic library code.

The DesignWare concepts of *synthetic module* and *implementation* closely correspond to the VHDL concepts of *entity* and *architecture*. An implementation can be viewed as an architectural realization of a synthetic module. For example, a ripple adder and a carry-lookahead adder are two possible implementations of an adder module. A DesignWare implementation can take the form of a netlist, a synthesizable RTL-level design description, or a simple hard macro.

1.4 How are DesignWare Building Block IP Used?

Users get access to DesignWare Building Block IP components through *component instantiation* and *operator inferencing*. In component instantiation, the user's HDL code explicitly includes an instance of a synthetic module. For operator inferencing, synthetic modules are automatically inferred from the presence of particular operators in the user's code.

1.5 Instantiation

The following example shows how a user might instantiate an adder in VHDL. The example instantiates the parameterized synthetic module `DW01_add`. The lines of code that refer to the adder module are shown in bold.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity adder is
    port(in1,in2:in STD_LOGIC_VECTOR(7 downto 0);
       carry_in:in STD_LOGIC;
       sum:out STD_LOGIC_VECTOR(7 downto 0);
       carry_out:out STD_LOGIC);
end adder;

architecture an_adder of adder is
    -- declare the component DW01_add
    component DW01_add
```

```
        generic (width: NATURAL);
        port(A,B: in STD_LOGIC_VECTOR(7 downto 0);
           CI: in STD_LOGIC;
           SUM: out STD_LOGIC_VECTOR(7 downto 0);
           CO: out STD_LOGIC);
end component;

begin
    -- instantiate DW01_add
    U1: DW01_add
        generic map(width => 8)
        port map(A => in1, B => in2, CI => carry_in, SUM => sum, CO =>
                  carry_out);

end an_adder;
```

The next example shows how a user might instantiate the same synthetic module in Verilog.

```
module adder(in1, in2, carry_in, sum, carry_out);

    parameter wordlength = 8;
    input [wordlength-1:0] in1, in2;
    input carry_in;  output [wordlength-1:0] sum;
    output carry_out;

    // instantiate DW01_add
    DW01_add #(wordlength)
        U1(in1,in2,carry_in,sum,carry_out);

endmodule
```

When the synthesis tool encounters the reference to `DW01_add`, it attempts to resolve the reference by looking for a synthetic module of that name in the synthetic libraries available. (The end user determines which libraries are available by setting the `synthetic_library` and `link_library` variables.)

When the synthesis tool finds the appropriate synthetic module, it looks for all the implementations that are associated with that module. Component designers associate implementations with modules by means of implementation declarations in the synthetic library source code.

Some implementations may be excluded from consideration because, for example, they have a fixed bit-width inappropriate to the current context, or because they are specific to a technology different from the user's target technology. Such exclusions are specified in advance by the component designer in synthetic library code.

All implementations that are not excluded in this way are candidates for inclusion in the user's design. The synthesis tools choose the implementation that best meets the user's constraints.

To characterize the implementations for comparison, the synthesis tool creates a pre-optimized model for each one, in the user's target technology. The timing and area characteristics of the models serve as the basis for implementation selection.

The chosen implementation is inserted, by default, as a level of hierarchy in the user's design, which is then mapped to the target technology and optimized.

1.6 Inference

An HDL operator—whether built into the language, like +, -, and *; or user-defined, like functions and procedures—can be linked to a synthetic operator. The linking mechanism is the `map_to_operator` pragma, which is included in the operator's HDL definition.

Note Only procedures that can be realized with purely combinational logic can be linked to synthetic operators.

Recall that a *synthetic operator* is an abstraction that makes it possible for the synthesis tools to perform arithmetic and resource-sharing optimizations before binding the operation to a particular synthetic module.

Operator inferencing occurs when the synthesis tool encounters an HDL operator whose definition contains a `map_to_operator` pragma. The tool finds the specified synthetic operator, inserts it into the user's design, and performs high-level optimizations on the resulting netlist.

The synthesis tool then performs implementation selection. It looks in the available synthetic libraries to determine which modules have bindings that link them to the synthetic operator. Implementations of all these modules are candidates for implementation selection.

Note Arithmetic optimization and resource sharing occur only during the user's first run of the compile command. Implementation selection, however, occurs (by default) on all subsequent compiles as well.

Synthetic operators for many of the built-in HDL operators are provided by Synopsys. These HDL operators include `+`, `-`, `*`, `<`, `>`, `<=`, `>=`, and operations defined by `if` and `case` statements. These synthetic operators, along with standard synthetic modules and bindings, are defined in the default synthetic library, `standard.sldb` ("Standard Synthetic Operators"). You can create your own synthetic operators by writing the appropriate synthetic library code ("Creating Synthetic Libraries: Basic" on page 47).

1.7 How Do I Create DesignWare Building Block IP Components?

DesignWare Developer is the tool you use to create DesignWare Building Block IP components from your design descriptions. DesignWare Developer is implemented as a collection of `dc_shell-t` commands and variables. Figure 1-2 shows the flow of information through DesignWare Developer.

Figure 1-2 DesignWare Developer

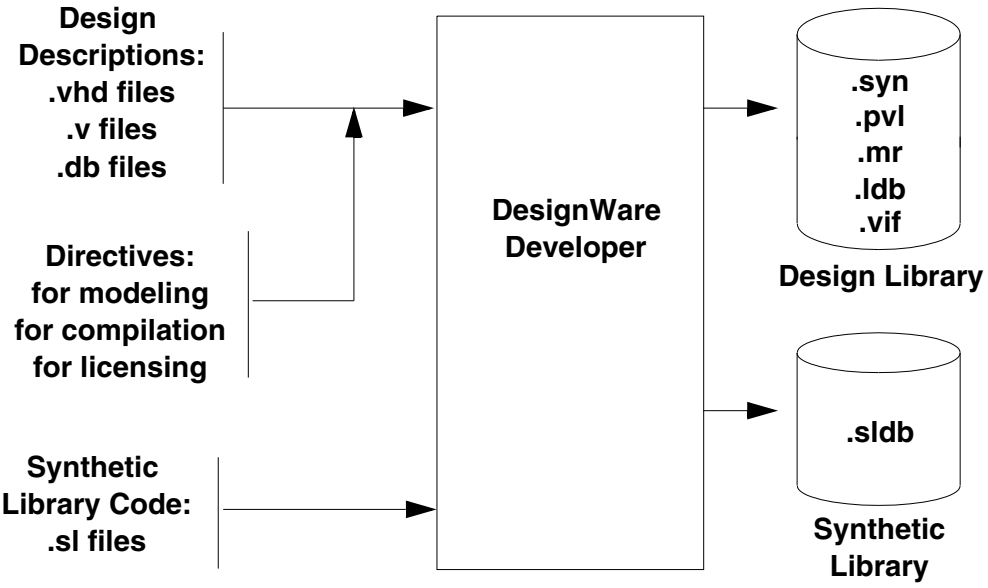


Figure 1-3 illustrates the general procedure for creating DesignWare Building Block IP components. The main steps are:

1. Create and verify your design.

   This book assumes you have already performed this step.

2. Apply directives for modeling, compilation, and licensing.

   For HDL design descriptions, you put your directives into `dc_shell-t` scripts embedded in the HDL code. For designs in other formats, you read the design into Design Compiler and apply directives by issuing the appropriate `dc_shell-t` commands.

   "Creating DesignWare Implementations" on page 39 discusses modeling and compilation directives.

3. Put your designs in a design library.

   A *design library* is a UNIX directory that contains design units that have been translated (analyzed) into a variety of intermediate formats that are directly usable by the Synopsys tools. You can create your own design libraries, or add your designs to existing ones.

   For HDL designs, you issue the `analyze` command to create the intermediate files and deposit them in a design library. For designs in other formats, you read the design into Design Compiler and (after applying the necessary directives) you issue the `write` command.

   "Creating DesignWare Implementations" on page 39 covers this step.

4. Write synthetic library code.

   You use a text editor to create the code that hooks your designs into Synopsys synthesis.

   You may simply want to connect your design to an existing synthetic module; in this way, you provide the module with an additional implementation. Or you may want to create a new synthetic

module with one or more implementations. You can even create your own synthetic operators and bind them to new or existing synthetic modules. (See Figure 1-1 on page 13.)

"Creating Synthetic Libraries: Basic" on page 47 covers the basics of declaring synthetic operators, modules, and implementations. "Creating Synthetic Libraries: Advanced" on page 65 discusses some advanced features.

5. Compile your synthetic library code.

   You use the `read_lib` and `write_lib` commands to compile your synthetic library code (`.sl` file) into a synthetic library (`.sldb` file). You can also augment an existing library.

   "Compiling and Verifying Synthetic Libraries" on page 73 covers this step.

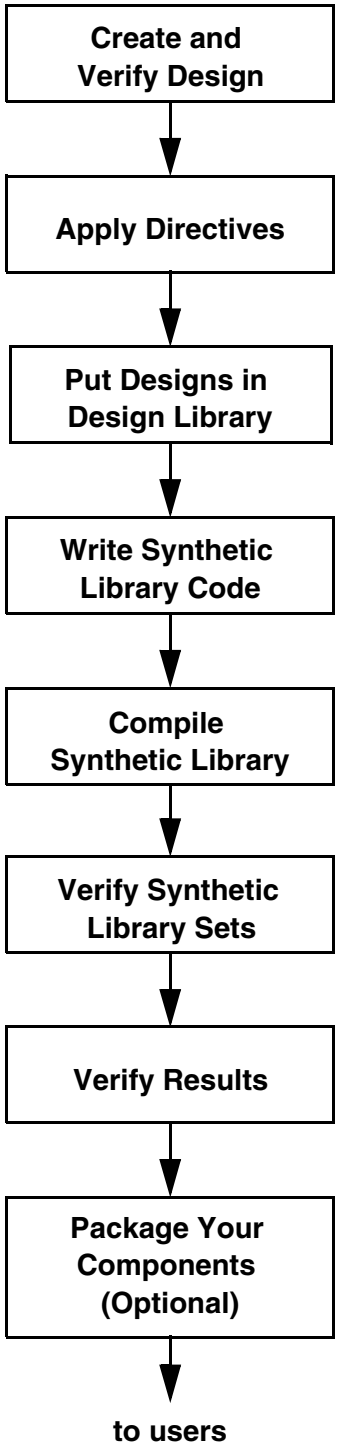
6. Verify your new DesignWare Building Block IP component.

   "Compiling and Verifying Synthetic Libraries" on page 73 tells you how to perform simple integrity tests for your component. You will also need to devise tests to verify that your implementations are selected when you expect them to be, that your designs are in fact protected from unlicensed access, and so on.

7. Package your DesignWare Building Block IP for distribution (optional).

   If you intend to distribute your components to third parties, there is a recommended directory structure for organizing the libraries you ship, and a recommended installation procedure; these are described in "Distributing DesignWare Building Block IP" on page 75.

Figure 1-3 Design Flow



2 Tutorial

This chapter takes you step-by-step through the process of creating DesignWare Building Block IP components from existing design data. The scenario is as follows: you want to provide an adder module that reports overflow in two's-complement addition. You have three implementations in hand: a ripple adder and a carry-lookahead adder coded in VHDL, and a technology-specific Verilog netlist for a handcrafted 8-bit implementation.

There are two series of exercises. In the first series, you work with the RTL-level VHDL implementations. In the second series, you learn how to create a technology-specific component using a Verilog netlist.

The parts of the tutorial are:

2.1 Setting Up the Tutorial

The files you need for the tutorial are included on the software distribution tape, in the directory `$SYNOPSYS/doc/dw/tutorial`. Copy this directory (and its contents, recursively) into your home directory.

2.2 Getting the Files You Need

1. Navigate to your home directory by typing: `cd`
2. Copy the tutorial files:

```
cp -r $SYNOPSYS/doc/dw/tutorial .
```

3. List the files:

```
ls tutorial
```

You should see the following files:

```
DWSL_addov.vhd
DWSL_addov_rpl.vhd
DWSL_addov_cla.vhd
```

The `.vhd` files contain VHDL design descriptions.

2.3 Creating a DesignWare Building Block IP Component from VHDL

In this series of exercises, you add directives to the (provided) VHDL source code for two adder implementations, and then analyze the code into a design library. You prepare a synthetic library description and create the synthetic library. As a final step, you check your implementations for consistency by creating timing models in the synthetic library cache.

2.3.1 Examine the Source Code

Example of Entity Declaration for Adder with Overflow Flag is the VHDL entity declaration for an adder with an overflow output flag. The entity is parameterizable for bit-width.

Example of Ripple Architecture and Example of Carry-Lookahead Architecture are VHDL definitions for ripple (`rpl`) and carry-lookahead (`cla`) adder architectures. The `cla` adder has two levels, and so can accommodate operands of up to 16 bits.

Both architectures use cells from the GTECH technology library. GTECH is a generic cell library that enables you to design technology-independent implementations at the cell (structural) level.

Both architectures use the VHDL `generate` statement. The `generate` statement is extremely useful for creating the regular gate structures that are often found in datapath elements. For parameterized regular structures, using structural VHDL with `generate` statements is often easier, and results in higher quality than the equivalent behavioral description.

If you are developing parameterized synthetic modules and implementations in Verilog (which has no equivalent for `generate`), you have to use behavioral descriptions.

The carry-lookahead architecture has an embedded `dc_shell-t` script. This script provides directives to the Synopsys synthesis software. You will add similar directives to the ripple architecture as part of the tutorial.

2.3.1.0.1 Example of Entity Declaration for Adder with Overflow Flag

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DWSL_addov is
    generic(width : POSITIVE);
    port(A,B : std_logic_vector(width-1 downto 0);
       CI : std_logic;
       SUM : out std_logic_vector(width-1 downto 0);
       OV, CO : out std_logic);
end DWSL_addov;
```

2.3.1.0.2 Example of Ripple Architecture

```
library IEEE, GTECH;
use IEEE.std_logic_1164.all;
use GTECH.GTECH_components.all;

architecture rpl of DWSL_addov is
    signal carry : std_logic_vector(width downto 0);

begin
    carry(0) <= CI;
    L1: for I in 0 to width-1 generate
       U1: GTECH_ADD_ABC PORT MAP(A(i), B(i), carry(i), SUM(i),
                     carry(i+1));
    end generate;
    CO <= carry(width);
    OV <= carry(width-1);
end rpl;
```

2.3.1.0.3 Example of Carry-Lookahead Architecture

```
library IEEE, GTECH;
use IEEE.std_logic_1164.all;
use GTECH.GTECH_components.all;

architecture cla of DWSL_addov is

-- pragma dc_tcl_script_begin
--   if { [get_attribute [current_design] {$width}] < 5 }  {
--      set_ungroup [current_design] "true" ;
--      }
-- pragma dc_tcl_script_end


    constant level1W : integer := width;
    signal level1P, level1G : std_logic_vector(level1W downto 0);
    signal level1C : std_logic_vector(level1W downto 0);

    constant level2W : integer := width/4;
    signal level2P, level2G : std_logic_vector(level2W downto 0);
    signal level2Ptemp, level2Gtemp : std_logic_vector(level1W downto 0);
    signal level2C : std_logic_vector(level2W downto 0);

    signal sumtemp : std_logic_vector(level1W-1 downto 0);

begin

-- compute level1 P and G
L1PG: for i in 0 to level1W-1 generate
    L1PG_U1: GTECH_OR2 port map(A(i), B(i), level1P(i));
    L1PG_U2: GTECH_AND2 port map(A(i), B(i), level1G(i));
end generate;

-- compute level1 C.
L1C: for i in 0 to level1W generate
    L1C_I1: if i mod 4 = 0 generate
       level1C(i) <= level2C(i/4);
end generate;

L1C_I2: if i mod 4 /= 0 generate
```

```
    L1C_U1: GTECH_AO21 port map(level1P(i-1), level1C(i-1), level1G(i-1),
                level1C(i));
    end generate;
end generate;

-- compute level2 P and G.
L2PG: if level2W /= 0 generate
    L2PG_L1: for i in 0 to level2W-1 generate

       level2Ptemp(i*4) <= level1P(i*4);
       level2Gtemp(i*4) <= level1G(i*4);

       L2PG_L2: for j in i*4+1 to i*4+3 generate
          L2PG_I1: if j <= level1W-1 generate
             L2PG_U1: GTECH_AND2 port map(level2Ptemp(j-1),
                level1P(j), level2Ptemp(j));
             L2PG_U2: GTECH_AO21 port map(level2Gtemp(j-1),
                level1P(j), level1G(j), level2Gtemp(j));
       end generate;
    end generate;

    L2PG_I2: if i*4+3 <= level1W-1 generate
       level2P(i) <= level2Ptemp(i*4+3);
       level2G(i) <= level2Gtemp(i*4+3);
    end generate;

    L2PG_I3: if i*4+3 > level1W-1 generate
       level2P(i) <= '0';
       level2G(i) <= '0';
    end generate;

    end generate;
end generate;

-- compute level2 C.
L2C: for i in 0 to level2W generate
    L2C_I1: if i mod 4 = 0 generate
       level2C(i) <= CI;
    end generate;

    L2C_I2: if i mod 4 /= 0 generate
       L2C_U1: GTECH_AO21 port map(level2P(i-1), level2C(i-1),
                   level2G(i-1), level2C(i));
    end generate;
end generate;

-- compute sums
FINAL_SUM: for i in 0 to level1W-1 generate
    FS_U1: GTECH_AND_NOT port map(level1P(i), level1G(i), sumtemp(i));
    FS_U2: GTECH_XOR2 port map(sumtemp(i), level1C(i), SUM(i));
end generate;

-- compute carry out and OV.
    OV <= level1C(level1W-1);
    CO <= level1C(level1W);
end cla;
```

2.3.2 Add Directives to the Source Code

Before you analyze your source code, you add a `dc_script` to each HDL source file. The directives in this script control the way Design Compiler models your implementations for implementation selection. You also include compile directives to control ungrouping.

2.3.2.1 Modeling Directives

When an instance of a synthetic module is included in a design (either by explicit instantiation or by inference), Design Compiler creates a pre-optimized timing model for every implementation of that module. The timing and area characteristics of the model serve as a basis for implementation selection.

At the time the timing models are created, the circuitry surrounding the synthetic module has not yet been mapped. Circuit characteristics that can affect the module's performance—fanouts and fanins, for example—are unknown.

You must therefore specify "typical" constraint and environment values for modeling by including the appropriate directives in a `dc_script` in your HDL source files. These directives are used for modeling only; for the final optimization in a target technology, the appropriate constraint and environment values are derived from context.

In the next exercises, you use modeling directives to assign loads and drives and to define optimization constraints for the ripple adder.

2.3.2.1.1 Optimization Constraints

Your next task is to set goals (constraints) for the modeling pre-optimization. You need to guide Design Compiler in the trade-offs it makes between circuit size and speed. Otherwise, your design may be poorly modeled and then unjustly passed over during implementation selection.

Unless you know your user's target technology, however, you cannot choose reasonable constraint values for either timing or area. As a first approximation, you specify either `max_area 0` (implying minimum area) or `set_max_delay 0` (implying maximum speed). These constraints cause your implementation to be modeled as small as it can be or as fast as it can be (respectively).

(More complex modeling is possible, using a two-pass pre-optimization. See "Creating DesignWare Implementations" on page 39.)

2.3.2.1.2 To set optimization constraints for modeling:

Add the following line to the VHDL file, immediately after the lines you added in the preceding exercise:

```
-- set_max_area 0
```

This directive causes the design to be pre-optimized for size, as is appropriate for a ripple adder. If Design Compiler pre-optimized this design for speed, the resulting model would probably be neither very small nor very fast. The implementation selection process would then never choose the ripple adder, even in contexts where size was of the essence.

2.3.2.2 Compile Directives

After modeling and implementation selection, Design Compiler inserts the chosen implementation into the user's netlist and compiles it in that context. Arrival and required times, input drives and output loads, are computed from the actual circuit environment of the implementation.

By default, the hierarchy of an implementation is preserved during compilation. You can override hierarchy preservation by using the `set_ungroup` command. This command allows the logic inside your design to be combined with the surrounding logic during compilation.

In the next exercise, you add a directive that ungroups the ripple adder for bit-widths of four or less.

2.3.2.2.1 To control ungrouping:

1. Add the following lines to the VHDL file, immediately after the line you added in the preceding exercise:

```
-- if { [get_attribute [current_design] {$width}] < 5 }  {
--     set_ungroup [current_design] "true" ;
-- }
-- pragma dc_tcl_script_end
```

   The directive uses the if statement from `dc_shell-t` and the `get_attribute` function. Its effect is to ungroup the design if the value of the generic width is less than five.

   get_attribute takes two arguments: a design name (here represented by the `dc_shell-t` variable `current_design`) and a generic name with a `$` at the beginning (as in `$width`). `get_attribute` returns the value of the specified generic in the specified design.

   The `dc_script_end` pragma ends the script. The following example shows the completed `dc_shell-t` script.

```
-- pragma dc_tcl_script_begin
-- set_max_area 0
-- if { [get_attribute [current_design] {$width}] < 5 }  {
--     set_ungroup [current_design] "true" ;
-- }
-- pragma dc_tcl_script_end
```

2. Save the file and quit the text editor.

2.3.3 Place the Designs in a Design Library

A design library is a UNIX directory that contains design units that have been translated (analyzed) into a variety of intermediate formats that are directly usable by the Synopsys tools.

You need to place the entity and architectures for each implementation in the appropriate design library. In the following exercises, you create a new design library, then use the analyze command to parse the VHDL source code into that library.

2.3.3.1 To create a design library:

1. Make sure you are in the right directory by typing

```
cd ~/tutorial
```

2. Create a new design library:

```
mkdir DWSL
```

   The DWSL directory will hold the analyzed design units.

It is also possible to add designs to an existing design library. You will do so later, in the Verilog exercises.

2.4 To analyze the VHDL design descriptions:

1. Invoke dc_shell-t:

```
dc_shell-t
```

2. Issue the command

```
define_design_lib DWSL -path ./DWSL
```

This command tells the tools that the logical library name DWSL is mapped to the UNIX directory `./DWSL`. `dc_shell-t` commands that refer to design libraries use the logical name.

3. Analyze VHDL with the command

```
analyze -format vhdl -lib DWSL [list DWSL_addov.vhd \
    DWSL_addov_cla.vhd DWSL_addov_rpl.vhd]
```

This command analyzes the VHDL code and populates the directory DWSL with analyzed design units.

4. Check the contents of the design library with the following command:

```
dc_shell> report_design_lib DWSL
```

This command generates a report on the contents of the DWSL library (as in the following example).

```
****************************************
Report : hdl libraries
Version: I-2013.12
Date   : <date>
****************************************
Contents of current design libraries
   DWSL (/<path>/DWSL)
      entity            : p    DWSL_ADDOV
      architecture      : d    DWSL_ADDOV(CLA)
      architecture      : m d  DWSL_ADDOV(RPL)

p --- This design has parameters.
m --- This architecture is the most recently analyzed.
d --- This file depends on design units that have been analyzed more recently.
```

5. Quit dc_shell-t

```
exit
```

You use a text editor to create the code that hooks your designs into Synopsys synthesis. The language you use is very similar to the Synopsys Library Compiler language (which is used for specifying technology libraries).

In the following exercises, you write code that creates a new synthetic library, adds a synthetic module (adder with overflow, `DWSL_addov`) to that library, and declares two implementations (`rpl` and `cla`) of the module.

2.5 To Define a New Synthetic Library:

1. Open a new file named DWSL.sl with a text editor.
2. Enter the following lines into the file:

```
library (DWSL.sldb) {<synthetic model declarations go here>
}
```

When you compile these lines in a later exercise, Design Compiler creates a new synthetic library with the name `DWSL.sldb`. The `.sldb` filename extension is required in the name of a synthetic library.

You enter synthetic module declarations between the braces of the `library` statement.

2.6 To add a synthetic module to the library:

1. Add the lines shown in boldface to the file `DWSL.sl`:

```
library (DWSL.sldb) {
    module (DWSL_addov) {
     }
}
```

The construct you have added is called a module group. It creates a synthetic module named `DWSL_addov` in the library `DWSL.sldb`.

For Foundation components based on VHDL design descriptions, synthetic module-implementation pairs correspond to the entity-architecture pairs of the VHDL code. The name of a synthetic module must match the name of the corresponding VHDL entity. Recall that `DWSL_addov` is in fact the name of the adder entity.

In the next few steps, you add information to the module group that is necessary to characterize the module `DWSL_addov`.

2. Add the lines shown in boldface:

```
library (DWSL.sldb) {
    module (DWSL_addov) {
    design_library : "DWSL";
    parameter (width) {
       hdl_parameter : TRUE;
    }
    }
}
```

You have attached an attribute (`design_library`) and a parameter group to the module.

The `design_library` attribute tells the synthesis tools where to find the actual design data for the module's implementations. In this case, the design library `DWSL` contains the analyzed VHDL design units.

The parameter group declares a width parameter for the module. This parameter corresponds to the `width` generic of the VHDL entity `DWSL_addov`, and is required to have the same name. In the next step of this exercise, you will use the `width` parameter to declare the bit-widths of the module's input and output ports.

Note The parameter name is case-sensitive; it must match the name in the VHDL generic declaration exactly.

The parameter group includes an attribute, `hdl_parameter`. Setting this attribute to `TRUE` tells the synthesis tools to look in HDL code for the value of the width parameter.

3. Add the lines shown in boldface:

```
library (DWSL.sldb) {
    module (DWSL_addov) {
    design_library : "DWSL";
    parameter (width) {
       hdl_parameter : TRUE;
    }
    pin (A) {
    direction : input;
    bit_width : "width";
    }
    pin (B) {
    direction : input;
    bit_width : "width";
    }
    pin (CI) {
    direction : input;
    bit_width : "1";
    }
    pin (SUM) {
    direction : output;
    bit_width : "width";
    }
    pin (OV) {
    direction : output;
    bit_width : "1";
    }
    pin (CO) {
    direction : output;
    bit_width : "1";
    }
    }
}
```

These are *pin groups*; they define the input and output ports of the module.

The pins of a synthetic module must match the ports of the corresponding VHDL entity in name (including case), direction, and bit-width.

Each pin group must have a `direction` attribute and a `bit_width` attribute. For the pins `A`, `B`, and `SUM`, the bit-width is defined in terms of the parameter `width`.

The definition of the module `DWSL_addov` is now complete. There are other optional attributes and groups that could be added; these options are discussed in "Creating Synthetic Libraries: Basic" on page 47 and "Creating Synthetic Libraries: Advanced" on page 65.

2.7 To Declare Implementations for the Module:

1. Add the following lines to the `DWSL_addov` module group, immediately after the last pin group. Use Completed Synthetic Library Description as a guide.

```
implementation (rpl) {
}
implementation (cla) {
    parameter (legal) {
       formula : "width < 17";
    }
}
```

These declarations inform the synthesis tools that there is an implementation of the synthetic module `DWSL_addov` named `rpl`, and one named `cla`. The tools will look for the actual design data for these implementations in the design library `DWSL` (thanks to the `design_library` parameter in the module group). For the tools to find the right designs in the design library, the implementation names must match the names of the corresponding VHDL architectures.

Since the `cla` adder has two levels, it can accommodate operands of up to 16 bits. The `legal` parameter tells the implementation selection mechanism that this architecture should be considered as an implementation only for adder instances of width 16 or less.

There are a number of optional attributes and groups that can be included in an implementation group; these options are discussed in "Creating Synthetic Libraries: Basic" on page 47 and "Creating Synthetic Libraries: Advanced" on page 65.

2. Save the file and quit the text editor.

2.7.0.0.1 Completed Synthetic Library Description

```
library (DWSL.sldb) {
    module (DWSL_addov) {
       design_library : "DWSL";
       parameter (width) {
          hdl_parameter : TRUE;
       }
       pin (A) {
       direction : input;
       bit_width : "width";
       }
       pin (B) {
       direction : input;
       bit_width : "width";
       }
       pin (CI) {
       direction : input;
       bit_width : "1";
       }
       pin (SUM) {
       direction : output;
       bit_width : "width";
       }
       pin (OV) {
       direction : output;
       bit_width : "1";
       }
       pin (CO) {
       direction : output;
       bit_width : "1";
       }
       implementation (rpl) {
       }
       implementation (cla) {
          parameter (legal) {
             formula : "width < 17";
          }
       }
    }
}
```

Your synthetic library source code is now complete. In the next section, you compile the code and verify that all the necessary connections between design library and synthetic library have been made correctly.

2.8 Compile and Check the Synthetic Library (.sldb)

You use the `dc_shell-t` commands `read_lib` and `write_lib` to compile synthetic library source code.

To compile your `.sl` file:

1. Invoke dc_shell-t by typing

   ```
   dc_shell-t
   ```

2. Enable the use of `write_lib` and `read_lib` commands

   ```
   enable_write_lib_mode
   ```

NOTE: This command gives a warning message that is expected: "Warning: This dc_shell session is for library operations only. You should not perform optimization on the design. Please exit when you are finished with the library"

3. Read your code into Design Compiler with the command

```
read_lib DWSL.sl
```

4. Write out the compiled synthetic library with the command

```
write_lib DWSL.sldb
```

The name of the output file specified on the command line (here, `DWSL.sldb`) must match the name declared for the synthetic library in the library statement of the source (`.sl`) file. (See the previous example.)

You can now generate a report on the new library with the `report_synlib` command, and you can create timing models for your implementations as a quick "sanity check."

To check your synthetic library:

1. Make the new library available to Design Compiler with the `dc_shell-t` command

```
set synthetic_library [concat DWSL.sldb $synthetic_library]
```

This command adds the new library to the beginning of the search path Design Compiler uses when it looks for synthetic-library information.

2. Check the contents of the library with the command

```
report_synlib DWSL.sldb
```

This command generates a report on the contents of the `DWSL.sldb` library (as in the following example).

```
****************************************
Report : library
Library: DWSL.sldb
Version: 1999.05-SI2
Date   : Mon Dec 7 14:37:40 1998
****************************************

Library Type: Synthetic
Tool Created: 1999.05
Date Created: Not Specified
Library Version: Not Specified

Synthetic Modules:

    Module
------------------------------------------------------------------------
    DWSL_addov       design_library: DWSL
                    HDL parameter: width


Module Pins:

    Attributes:
       c - clock_pin
                              Default Stall  Pin
```

```
    Module      Pins     Dir  Width      Value   Pin Attributes
-------------------------------------------------------------------------
    DWSL_addov       A in   width
                B  in   width
                CI in   1
                SUM out  width
                OV out  1
                CO out  1

Module Implementations:

    Attributes/Parameters:
       v - verify_only
       V - verification implementation
       u - dont_use
       r - regular_licenses
       l - limited_licenses
       d - design_library
       s - priority_set_id
       p - priority
       leg - legal

    Module           Implementations      Attributes/Parameters
    ---------------------------------------------------------------------------
    DWSL_addov       rpl

          cla                  leg = "width < 17"


Module Bindings:

Module              Binding
-------------------------------------------------------------------------
```

3. Make the necessary libraries available to `dc_shell-t` by issuing the following sequence of commands:

```
define_design_lib DWSL -path ./DWSL
set target_library class.db
set link_library class.db
```

4. Verify that Design Compiler can at least create timing models for your implementations by issuing the command

```
create_cache -module DWSL_addov -parameters [list {width = 4} \
{width = 8}] -report
```

Recall that for the purpose of implementation selection, the synthesis tools create models of all contending implementations. It stores these models in a UNIX directory called the *synthetic library cache*. When a synthesis tool needs a model that is already in the cache, it does not have to create the model again.

The `create_cache` command allows users of Foundation parts to control the initial population of the cache. The `-parameters` option tells the synthesis tools what values to use for module parameters. The `-report` option generates a report that shows details about the models created (as in the following example).

```
Implementation: 'rpl'
      Parameters: {  (width =   8.00)  }
      Completed.
Information: Updating design information... (UID-85)

  Point                                    Incr       Path
-----------------------------------------------------------
  input external delay                     0.00       0.00 f
  A_0 (in)                                 0.00       0.00 f
  ...
  SUM_7 (out)                             16.55      16.55 r
-----------------------------------------------------------
```

You now have an adder-with-overflow module and two working implementations. When a user instantiates the entity `DWSL_addov` in his or her design, Design Compiler will model both implementations and determine which is better in the context of the user's circuit.

You are now in a position to add additional implementations. You do so in the next series of exercises.

2.9 Creating a Technology-Specific DesignWare Component

In this series of exercises, you start with Verilog code for a handcrafted 8-bit adder implementation (with overflow). The 8-bit adder is realized in the technology library `class`. The `class` library does not exist in silicon; it is simply a teaching tool developed at Synopsys.

In the exercises, you read the code into Design Analyzer, apply directives, and write the design out to the design library. You add a new implementation declaration to the synthetic library you created in the last series of exercises (`DWSL.sldb`). As a final step, you check your implementation for consistency by creating a timing model in the synthetic library cache.

2.9.1 Create a Synopsys Design Database (.db)

Design data for DesignWare implementations can be specified in any input format accepted by the Synopsys synthesis tools. For non-HDL formats, your first step is to read the design into the Synopsys environment.

2.10 To Create a Synopsys Design Database:

Issue the `dc_shell-t` command

```
read_file -format verilog <verilog_file_name>
```

This command reads the Verilog design description into Design Compiler. The design now resides in memory in `.db` format.

2.11 Apply Directives

You apply directives to a non-HDL design by issuing commands directly to `dc_shell-t`. (Recall that for HDL designs, you embed the directives in a `dc_shell-t` script inside the HDL code.)

2.12 To apply directives:

1. Rename the design so that it will be recognized as an implementation of the synthetic module `DWSL_addov`:

```
rename_design new_adder DWSL_addov__proprietary
```

To be connected to the declarations in a synthetic library, a non-HDL design unit must have a name of the form

module_name__implementation_name

With the `rename` command you have just issued, you have ensured that the 8-bit adder will be recognized as an implementation of the synthetic module `DWSL_addov`. When you create the implementation declaration for this design, you must use the implementation name `proprietary`.

2. Prevent Design Compiler from attempting to optimize your handcrafted design by issuing the command

```
set_dont_touch DWSL_addov__proprietary
```

This command prevents Design Compiler from touching your design. When `DWSL_addov__proprietary` is selected as the best implementation for an instance of `DWSL_addov`, `DWSL_addov__proprietary` is included in the user's netlist and undergoes no further processing.

3. Provide Design Compiler with linking information by issuing the command

```
set_local_link_library class.db
```

The Verilog code for the adder contains references to cells in the `class` library. Design Compiler needs to know where to look for these cells when it elaborates your design. The `set_local_link_library` command supplies this information.

2.13 Place the Designs in a Design Library

You now write the design to disk, adding it to the design library you created in the last series of exercises (`DWSL`).

2.14 To Write Out the Design:

1. Issue the command

```
write -library DWSL
```

This command adds the design (filename `DWSL_addov__proprietary.ldb`) to the design library `DWSL`.

2. Check the contents of the design library with the command

```
report_design_lib DWSL
```

The report should now show three implementations in the library.

2.15 Write a Synthetic Library Description (.sl)

You now write `.sl` code to attach the 8-bit adder to the existing synthetic module `DWSL_addov`. You could in principle edit the file `DWSL.sl` that you created earlier and add a third implementation declaration. You could then recompile the synthetic library.

There may be times, however, when you do not have access to the `.sl` source code for a module, but you nonetheless want to add an implementation to it. DesignWare Developer allows you to declare an

implementation outside the scope of the associated module. You can then compile the external declaration into the existing library (`.sldb` file).

2.16 To declare the proprietary implementation:

1. Open a new file named `proprietary.sl` with a text editor.
2. Enter the following lines into the file:

```
implementation (proprietary,DWSL_addov) {
    technology : class.db;
    parameter (legal) {
       formula : "width == 8";
    }
}
```

The implementation name—as discussed earlier—is `proprietary`. Since this declaration occurs outside the scope of a synthetic module definition, the module name (in this case, `DWSL_addov`) must also be provided.

The `proprietary` implementation is technology-specific. Setting the `technology` parameter in this way tells Design Compiler to consider `proprietary` for implementation selection only if the user's target technology is `class`.

The `proprietary` implementation is hard-coded for 8-bit ports. The legal parameter tells Design Compiler to consider `proprietary` only for those instances of `DWSL_addov` whose `width` parameter equals 8.

2.17 Compile and Check the Synthetic Library

You can now perform an incremental compilation that adds the new implementation to `DWSL.sldb`.

2.18 To compile your new .sl file:

1. Read in the existing library with the `dc_shell-t` command

```
read DWSL.sldb
```

2. Compile the new implementation into the library with the command

```
update_lib DWSL.sldb proprietary.sl
```


Do not use `update_lib` on the synthetic library *standard*`.sldb` because parts in *standard*`.sldb` are licensed differently from user parts.

3. Write out updated library with the command

```
write_lib DWSL.sldb
```

You can generate a report on the updated library with the `report_synlib` command, and you can create timing models for all three implementations.

2.19 To Check Your Synthetic Library:

1. Check the contents of the library with the command

```
report_synlib DWSL.sldb
```

The report should now include the external implementation `proprietary` (Se the following example).

2. Verify that Design Compiler can at least create timing models for your implementations by issuing the command

```
create_cache -module DWSL_addov -parameters [list {width = 4} \
{width = 8}] -report
```

This command generates models of both specified widths for the `rpl` and `cla` implementations. For `proprietary`, only an 8-bit model is created.

3. Quit dc_shell-t:

```
exit
```

You now have three implementations of the adder-with-overflow module. When a user instantiates the entity `DWSL_addov` in his or her design, Design Compiler will model the valid implementations and determine which is best in the context of the user's circuit.

```
External Implementations:

 Attributes/Parameters:
    v - verify_only
    u - dont_use
    r - regular_licenses
    l - limited_licenses
    d - design_library
    s - priority_set_id
    p - priority
    leg - legal

Module  Implementations Attributes/Parameters
--------------------------------------------------------------------------
DWSL_addov proprietary * leg = "width == 8"
```

3 Creating DesignWare Implementations

This chapter tells you how to prepare your design descriptions for use as DesignWare implementations. The style of your source descriptions can vary from a technology-specific netlist or hard macro that will not be altered by synthesis, up to a full hierarchical HDL description of a parameterizable, optimizable design.

You turn your "raw" design descriptions into DesignWare implementations by adding information that helps guide the Synopsys tools during implementation selection and optimization. You then place the implementations in a design library, making them available for use. The steps are as follows:

3.1 Creating Implementations: DesignWare Rules

When you create your implementation source code, there are certain DesignWare rules you should be aware of. These rules pertain to

- Design unit names—The rules require consistency between the names of design units in your design library and the name references used in your synthetic library code.
- Parameterization—The rules determine the type of parameters (*generics*, in VHDL; *parameters*, in Verilog) that are allowed and the circumstances under which you can (or must) include parameters in your design description.

The exact nature of the rules depends on the kind of design description you are writing.

3.1.1 VHDL Implementations

The VHDL concepts of *entity* and *architecture* correspond directly to the DesignWare concepts of *synthetic modules* and *implementations*. The names of corresponding objects must match, according to the following rule:

> When a DesignWare implementation is based on VHDL source code, the architecture name must match the implementation name, and the entity name must match the synthetic module name.

The following example shows VHDL declarations and the corresponding `.sl` declarations. The entity `SL_addov` has two architectures, `rpl` and `cla`. To turn these design units into DesignWare, the naming rule requires that rpl and cla be declared in the synthetic library as implementations of the module `SL_addov` (as shown).

Note The syntax for declaring synthetic modules and implementations in synthetic library code is described in detail in "Creating Synthetic Libraries: Basic" on page 47.

VHDL generics correspond to DesignWare parameters. The correspondence is governed by the following rule:

> Generics in your VHDL source code must be of an integer type, and their names must match the names of parameters in the corresponding synthetic module declaration.

In the following example, entity `SL_addov` has a generic `width`, of type `INTEGER`. The module `SL_addov` therefore has a `width` parameter declared in its synthetic library description. Note that the `hdl_parameter` attribute of `width` is set to `TRUE`: this indicates that `width` is declared in HDL code. Finally, the bit-widths, names, and directions of the entity ports must match those declared for the module ports.

```
VHDL:

entity SL_addov is
    generic(width : POSITIVE);
    port(A,B : std_logic_vector(width-1 downto 0);
        CI : std_logic;
        SUM : out std_logic_vector(width-1 downto 0);
        OV, CO : out std_logic);
end SL_addov;

architecture rpl of SL_addov is
begin
    -- description goes here
end rpl;

architecture cla of SL_addov is
begin
    -- description goes here
end cla;

.sl:
module (SL_addov) {
    design_library : "SL";
        parameter (width) {
            hdl_parameter : TRUE;
        }
        pin (A) {
        direction : input;
        bit_width : "width";
        }
        pin (B) {
        direction : input;
        bit_width : "width";
        }
        pin (CI) {
        direction : input;
```

```
        bit_width : "1";
        }
        pin (SUM) {
        direction : output;
        bit_width : "width";
        }
        pin (OV) {
        direction : output;
        bit_width : "1";
        }
        pin (CO) {
        direction : output;
        bit_width : "1";
        }
        implementation (rpl) {
        }
        implementation (cla) {
        }
}
```

3.1.2 Verilog Implementations

Verilog does not use architectures. Instead, each implementation of a synthetic module is a separate Verilog module. Names are related by the following rule:

> When a DesignWare implementation is based on Verilog source code, the Verilog module name must be of the form *synthetic_module_name__implementation_name*.

For example, the Verilog module for a `ripple` implementation of the synthetic module `SL_addov` would be called `SL_addov__ripple`.

Verilog *parameters* correspond to a synthetic module parameters; their names must match (as in the rule for VHDL), and the `hdl_parameter` attribute of the synthetic module parameter must be set to `TRUE`.

The bit-widths, names, and directions of the Verilog module ports must match those declared for the synthetic module ports.

3.1.3 .db Implementations

You can describe your implementation in any format that Synopsys supports and then convert the design to `.db` format by reading it into Design Compiler.

Each implementation of a synthetic module requires a separate .db file. Names are related by the following rule:

> When a DesignWare implementation is based on a design in .db format, the design name must be of the form synthetic_module_name__implementation_name.

For example, the `.db` design for the 8-bit implementation of the synthetic module `SL_addov` is called `SL_addov__width8`. (See "Tutorial" on page 21)

If necessary, you can use the `rename_design` command to set a new name on your design after you have read it into Design Compiler.

The bit-width, names, and directions of the `.db` design ports must match those declared for the synthetic module ports.

Note Because gate-level.db designs cannot be parameterized, they require special treatment when associated with a synthetic module that uses an `hdl_parameter`. The implementation declaration ("Creating Synthetic Libraries: Basic" on page 47) of the gate-level design must use a `legal` parameter group to specify which value of the HDL parameter is valid for the design. (See the Verilog example in the tutorial of "Tutorial" on page 21.)

3.2 Adding Modeling Directives

When a module instance is included in a design (either by explicit instantiation or by inference), Design Compiler may create a pre-optimized timing model for every valid implementation of that module. The timing and area characteristics of the model serve as a basis for implementation selection.

At the time implementation selection occurs, the circuitry surrounding the synthetic module may or may not be mapped, depending on whether timing constraints exist. Some circuit characteristics that can affect the module's performance—fanouts, for example—may be unknown.

You should therefore specify "typical" constraint and environment values for modeling by including the appropriate directives in a `dc_script` in your HDL source files. These directives are used for modeling only; for optimization in a target technology, the appropriate constraint and environment values are derived from context.

You use modeling directives to assign estimated loads and drives to the ports of the implementation, and to specify optimization goals—whether the implementation should be pre-optimized for size or for speed.

3.2.1 Step 1. Assigning Loads and Drives

This step is no longer needed and has been deprecated.

3.2.2 Step 2. Setting Constraints

Your next task is to set goals (constraints) for the modeling pre-optimization. You need to guide Design Compiler in the trade-offs it makes between circuit size and speed.

Failure to set constraints can result in sub-optimal implementation selection. In particular, if you do not include modeling directives, Design Compiler will model for minimum area by default. Without guidance, Design Compiler will pre-optimize your fast implementations for size, and may well arrive at models that are neither very small nor very fast. In such a case, your implementation is likely to be unjustly passed over during implementation selection.

Unless you know your user's target technology, however, you cannot choose reasonable constraint values for either timing or area. As a first approximation, you specify either `max_area 0` (implying minimum area) or `max_delay 0` (implying maximum speed). These constraints cause your implementation to be modeled as small as it can be or as fast as it can be (respectively).

3.3 Adding Compilation Directives

After modeling and implementation selection, Design Compiler inserts the chosen implementation into the user's netlist and compiles it in that context. Arrival and required times, input drives and output loads, are computed from the actual circuit environment of the implementation.

To apply `dc_shell-t` compilation directives to an HDL implementation, include the directives in a `dc_script` in your source file. For a design in some other format, read the design into Design Compiler and issue the appropriate `dc_shell-t` commands.

Compilation directives include `dont_touch`, `set_boundary_optimization`, `set_structure`, `set_flatten`, and `set_ungroup`. You use these directives to control the in-context optimization of your designs.

For example, if one of your implementations is a handcrafted gate-level netlist, you may want to apply a `dont_touch` directive to the design, to prevent Design Compiler from attempting to optimize it. See the *Design Compiler Family Reference Manual* for a full discussion of these directives.

Some compilation directives require no object lists. For example, the `set_structure` directive runs on the current design by default:

```
set_structure false /* defaults to current_design */
```

Other compilation directives, such as `set_ungroup`, `dont_touch`, and `set_boundary_optimization`, require an object list to be defined. If you want to use these directives on the current design, use the `current_design` variable as your object list:

```
set_boundary_optimization current_design false
```

3.4 Adding Hierarchy-Control Directives

Implementations can be hierarchical; in particular, they can contain instances of

- Technology library cells
- Generic technology cells (from the GTECH library)
- Synthetic modules

For example, an implementation of a multiplier module can contain several instances of an adder module. Implementations can also contain HDL operators that are mapped to synthetic operators.

An implementation cannot contain any instance of a design that is not in a technology library or a synthetic library.

When your implementation is hierarchical, you must set up a linking path so that Design Compiler can find the components instantiated in the design. You can also (optionally) control whether Design Compiler ungroups your design before optimization, and whether it performs resource sharing on the synthetic operators and modules (inferred or instantiated) in the design.

3.4.1 Step 1. Set Up the Link Path

When elaborating a hierarchical design, the synthesis tools normally search the libraries listed in the `link_library` variable to resolve references to subblocks. However, you cannot control how the end user defines the `link_library` variable. If your DesignWare component is hierarchical, you must "build in" the necessary linking information. The end-user's setting of `link_library` is ignored.

References to subblocks in a hierarchical DesignWare implementation are resolved by a search through libraries in the following order:

1. Designs and libraries specified by the `set_local_link_library` command
2. standard.sldb
3. The synthetic library that contains the current module
4. The synthetic library that contains the current implementation
5. The generic technology library (`gtech.db`)

You can set `local_link_library` variable with the directive `set_local_link_library`. Use the `set_local_link_library` directive to include technology libraries and synthetic libraries not linked by default; you can also use the directive to define your own link order.

For technology-specific implementations, you must include the target technology library in the search path by using the `set_local_link_library` command. Also, for implementations that have a hierarchy of synthetic library parts, you must include the technology library in the `set_local_link_library` command for all levels of the synthetic library hierarchy, from the first instance of the technology-specific part to the top-level synthetic library part.

In the following example, the Synthetic Library `my_synlib.sldb` is added to the linking path of the implementation impl1 with the `set_local_link_library` directive. The directive is shown in bold.

```
architecture impl1 of adder is
-- pragma dc_tcl_script_begin
-- set_local_link_library { "my_synlib.sldb" }
-- pragma dc_tcl_script_end
begin
-- Description goes here
end impl1;
```

To resolve references to libraries—like the reference in the above example to `my_synlib.sldb`—Design Compiler normally searches the directories listed in the `search_path` variable. However, you cannot control how the end user defines the `search_path` variable.

References to libraries that occur within a DesignWare implementation are resolved by a search through directories in the following order:

1. The directory of the `.sldb` file that defines the module
2. The directory of the `.sldb` file that defines the implementation
3. The directory of the current `target_library`

3.4.2 Step 2. Remove Hierarchy after Compile (Optional)

By default, the internal hierarchy of an implementation is preserved during compilation. You can override hierarchy preservation by using the `set_ungroup` command on selected cells within the implementation. You can also force your entire synthetic module to be ungrouped into its containing design (the end user's design); use the `set_ungroup` command on `current_design` to accomplish this.

You can control the conditions under which implementations are ungrouped into their containing designs. For example, if you want implementations to be ungrouped automatically if they are four bits wide or smaller, you can use the if statement from `dc_shell-t` with the `get_attribute` function (shown in **`bold`** in the following example).

`get_attribute` takes two arguments: a design name (here represented by the `dc_shell-t` variable `current_design`) and a generic name with a dollar sign ($) at the beginning (as in `$width`). `get_attribute` returns the value of the specified VHDL generic or Verilog parameter in the specified design.

```
architecture impl1 of adder is
-- if { [get_attribute [current_design] {$width}] < 5 }  {
--     set_ungroup [current_design] "true" ;
-- }
begin
-- Description goes here
end impl1;
```

In the current compile flow, DC automatically ungroups sub DW component based on certain criteria. If a DW design is small enough (default is < 20 cell count); or has certain percentage (default is >25%) of constant inputs, the design will be ungrouped, even without the embedded set_ungroup command.

3.5 Placing Implementations in a Design Library

Before your designs can be used or tested, they have to be *analyzed*—translated into a Synopsys intermediate format—and placed in a design library. You can either create a new design library or add your implementations to an existing library.

3.5.1 Creating a Design Library

Recall that a *design library* is just a UNIX directory that contains analyzed design units. You create the directory with the UNIX `mkdir` command, then use the `dc_shell-t` command `define_design_lib` (or the `.synopsys_sim.setup` file) to associate the directory with a library name. You use the *library name* (typically a valid HDL identifier) to refer to the library in subsequent commands.

3.5.2 Adding Implementations to a Design Library

The method you use to `analyze` your designs and place them in a design library depends on the nature of your source design descriptions.

3.5.2.1 HDL Implementations

You use the `analyze` command to parse HDL source code and to place the resulting intermediate file(s) in the appropriate design library.

For example, to place the source file `my_source.vhd` in the design library `MY_LIB` (which resides in `/usr/libraries/my_lib`), enter the following commands:

```
define_design_lib MY_LIB -path /usr/libraries/my_lib
analyze -library MY_LIB -f vhdl my_source.vhd
```

To place the source file `my_source.v` in the design library `MY_LIB` (which resides in `/usr/libraries/my_lib`), enter the following commands:

```
define_design_lib MY_LIB -path /usr/libraries/my_lib
analyze -library MY_LIB -f verilog my_source.v
```

3.5.2.2 Other Implementations

After reading an implementation into Design Compiler and applying the necessary directives, you write the design out to a design library by using the `write` command with the `-library` flag.

The following example illustrates the procedure for reading in and writing out an implementation of an adder module defined in Verilog format.

```
read -f vlog my_design.v
set_structure false
rename_design my_design adder__impl1
define_design_lib MY_LIB -path /usr/libraries/my_lib
write -library MY_LIB
```

The procedure for reading in and writing out an implementation is:

1. The implementation file (named `my_design.v`) is read in with the `read` command.
2. The `set_structure` false command turns off structuring in the already well structured synthetic part.
3. The `rename_design` command sets the name of the implementation to the appropriate format (`module__implementation`).
4. The `define_design_lib` command maps the name of the design library `MY_LIB` to a UNIX directory.
5. The `write` command places the db designs in the appropriate design library (`MY_LIB`).

4
Creating Synthetic Libraries: Basic

Synthetic libraries link your designs into the Synopsys high-level design tools. They provide the "glue" that associates implementations with modules, and modules with synthetic operators.

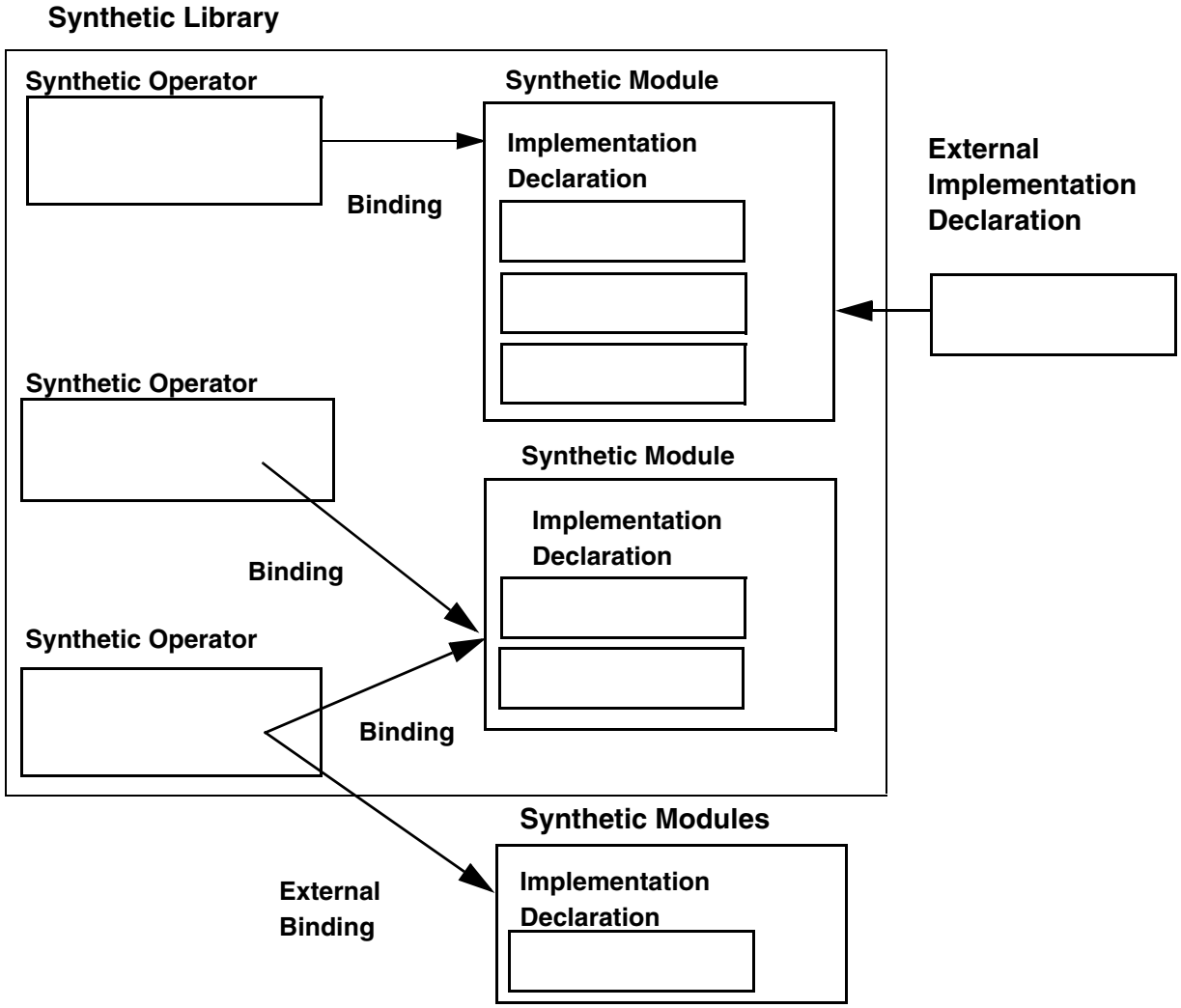
To create a synthetic library, you write code in a simple language that resembles the language used in Library Compiler to describe technology libraries. You then compile the code by reading it into the Synopsys environment and writing it out as a library.

This chapter covers the following topics:

4.1 Anatomy of a Synthetic Library

Figure 4-1 illustrates the basic structure of a synthetic library. For definitions of the terms that appear in the diagram, see "DesignWare Concepts" on page 11.

Figure 4-1 Synthetic Library Structure



You write code that defines this structure, or part of it. Depending on your needs, you may simply want to connect your design to a module that already exists in some synthetic library; in this way, you provide the module with an additional implementation. Or you may want to create a new module with one or more implementations. You can even create your own synthetic operators and bind them to new or existing modules.

4.1.1 For All Components

Providing declarations for your implementations is the minimum necessary step for turning them into DesignWare Building Block IP components. *Implementation declarations* connect your analyzed design units (residing in a design library) with synthetic modules. Users of your DesignWare Building Block IP components can explicitly instantiate any synthetic module in their HDL code; the synthesis tools then

model the declared implementations for that module and select the best one, given the user's overall optimization goals.

Many synthetic modules are available in the library `standard.sldb`. You can list the modules in any synthetic library you have access to by using the `report_synlib` command in `dc_shell-t`.

If none of the existing synthetic modules fits your purpose, you can create a new one by writing your own *module declaration*.

As the diagram shows, an implementation declaration can be made either within the scope of a module declaration, or externally. External implementation declarations allow you to connect your implementations to the modules in an existing library, even if you do not have access to the source code for that library.

4.1.2 For Inferable Components

If your DesignWare Building Block IP component is combinational, you can automate your user's task further: you can establish links that make it possible for the synthesis tools to *infer* instances of your synthetic module from the presence of a particular HDL operator in your user's code.

You can infer sequential parts by using Behavioral Compiler.

To make your component inferable, you need two things:

- A *binding declaration* that associates your synthetic module with a synthetic operator.
- A `map_to_operator` pragma in the HDL definition of the appropriate HDL operator. This pragma maps the HDL operator to a synthetic operator.

As the diagram shows, bindings can be declared either within the scope of a module declaration, or externally.

Many synthetic operators are available in the library `standard.sldb`. Use the `report_synlib` command to find out about the synthetic operators that are available to you.

If none of the existing synthetic operators meet your needs, you can create one by writing an *operator declaration*. An operator declaration cannot stand on its own: if you declare your own operator(s), you must do so within the scope of a new or existing *library declaration*.

The remainder of this chapter tells you how to write synthetic library code, and how to use the `map_to_operator` pragma.

4.2 Writing Synthetic Library Code

You write your synthetic library source code in a file whose name has a `.sl` extension. The syntax of synthetic library code is similar to that of Synopsys technology library code. The following sections describe the syntax and semantics of the most important language constructs. Additional constructs are covered in "Creating Synthetic Libraries: Advanced" on page 65 and "Compiling and Verifying Synthetic Libraries" on page 73.

4.2.1 Basic Syntax

Synthetic library syntax consists of *groups*, *attribute statements*, and *comments*. You generally enter one statement per line. You can continue a statement to the next line by ending the current line with a backslash (\).

4.2.1.1 Groups

A *group* is a named collection of statements. In synthetic libraries, groups are used to define libraries, modules, implementations, bindings, operators, parameters, and pins. Braces ({}) delimit the scope of the group. Groups can be nested; the nesting of groups reflects the hierarchical structure of the synthetic library ("Synthetic Library Structure" on page 48).

The general syntax of a group statement is

```
group_name (name [, name]) {
    ... statements ...
}
```

`group_name` is `library`, `module`, `implementation`, `binding`, `operator`, `parameter`, or `pin`, and the first `name` is a string that identifies the group. The second `name` is required by some groups; when present, it typically refers to some other group. Check the syntax of each group statement to verify whether `name` is required, optional, or null. You must include the group name(s) and the { symbol on the same line.

The following example shows the `pin` group `A`.

```
pin(A) {
   ... pin group statements ...
}
```

4.2.1.2 Attribute Statements

An *attribute* statement specifies some characteristic of the group that contains it. The following example adds a `direction` attribute with the value `output` to the `pin` group from the previous example.

```
pin(A) {
    direction : output ;
}
```

The syntax of an attribute statement is

```
attribute_name : attribute_value ;
```

You must separate the attribute name from the attribute value with a space, followed by a colon, followed by another space. The attribute statement must be on a single line.

4.2.1.2.1 Formulas

A variation on the basic attribute syntax is required by the `formula` attribute. This attribute is used in a *parameter group* to specify a value for the parameter. Parameter groups can be used inside module and implementation groups; the values assigned to a parameter typically control some facet of synthetic module building or modeling.

The syntax of the `formula` attribute is

```
formula : "expression" ;
```

`expression` must be enclosed in quotes, as shown. The expression can be a numerical or string constant, or it can be a complex expression written in terms of other parameters defined in the current group.

The following example assigns the value `X + Y` to the parameter `A`.

```
parameter (X) {
    ...
}
parameter (Y) {
    ...
}
parameter (A) {
    formula : "X + Y" ;
}
```

Built-in functions are provided for constructing complex expressions. In the above example, "`X + Y`" represents the sum of the values of the parameters `X` and `Y`.

The functions available for building expressions include the infix operators `+`, `-`, `*`, `/`, `%`, `^`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `!`, `&&`, and `||`. Table 4-1 shows the available infix operators and their precedence.

Table 4-1 Operators and Their Precedence

Operator	Function	Precedence
||	OR	lowest
&&	AND	
= = ! = < > < = > =	comparison	
+ -	addition, subtraction	
* / %	multiplication, division, mod	
! ^	NOT, exponentiation	highest

Several prefix functions are also available for use in formulas. For example, the `width` function returns the width of the specified module port (refer to "Defining Modules" on page 54 for details). Table 4-2 shows the available prefix functions.

Table 4-2 Functions for Use in Formulas

Function	Argument(s)	Returns
max	comma-separated list of numbers (integer or float)	larger argument
min	comma-separated list of numbers (integer or float)	smaller argument
log2	number (integer or float)	base-two logarithm of floating-point argument; bit-width (ceiling of base-two log) for integer argument
width[a]	name of module port	bit-width of port
size	name of implementation, in single quotes	area estimate for implementation

a. This function is only valid when used in parameter groups immediately defined within the module.

Formulas can include conditionals. The conditional expression

```
condition ? value_1 : value_2
```

has value `value_1` if `condition` is true (nonzero), and `value_2` otherwise.

The following example specifies the syntax of formulas in detail.

```
formula : "expression"
        ;
expression : ( expression )
        | function_name ( expression { , expression } )
        | expression infix_operator expression
        | expression ? expression : expression
        | constant
        ;
constant: digits          /*integer*/
        | 'characters'    /*string*/
        | digits.digits   /*floating point*/
        ;
```

4.2.1.3 Comments

Comments explain library entries. Comments are not a mandatory part of library syntax; however, liberal use of comments helps you to document your specifications. You must enclose comments between the delimiters /* and */ as shown below:

```
/* This is a one-line comment. */

/* This is a
  multi-line comment. */
```

DesignWare Developer ignores all characters between the diameters.

4.2.2 Defining Libraries

A synthetic `library` group is similar to a technology `library` group. The `library` group contains `operator`, `module`, and `implementation` groups. By convention, the name of the library ends with the `.sldb` extension. This extension identifies the library as a synthetic library and ensures that the library name and the final library filename are the same. The following example is the group statement for the synthetic library `my_synlib.sldb`.

```
library ( my_synlib.sldb ) {
    /*
    operator, module, and implementation groups go here
    */
}
```

4.2.3 Defining Operators

You define new operators by adding an operator group to your synthetic library. Within the `operator` group, you define the pins of the operator. You can also specify whether sign-extension or zero-extension is performed on input data, and which pairs of input variables can be swapped.

By convention, operator names end in the letters `_OP`.

The following example is the declaration for an operator that performs signed addition with carry-in. It contains `data_class` and `permutable_inputs` attribute statements, and several pin groups.

```
operator (SIGNED_ADD_CI_OP) {
    data_class : "signed";
    permutable_inputs : "A B";
    pin(A) {
       direction : input;
    }
    pin(B) {
       direction : input;
    }
    pin(CI) {
       direction : input;
    }
    pin(Z) {
       direction : output;
    }
}
```

The operator `SIGNED_ADD_CI_OP` has three input values (`A`, `B`, and `CI`) and one output value (`Z`). According to the `data_class` attribute, input data is sign-extended, if necessary. The `permutable_inputs` attribute indicates that the input values `A` and `B` of the operator are interchangeable.

To make your synthetic operator inferable (and therefore useful), you must not only declare it; you must also connect it to one or more HDL operators. See "Mapping HDL Operators to Your Components" on page 62.

4.2.3.1 Operator Attributes

There are two optional attributes commonly used in operator groups, `data_class` and `permutable_inputs`.

4.2.3.1.1 data_class Attribute

The `data_class` attribute on an operator describes how (and whether) to extend input data. `data_class` can have the values "`signed`" or "`unsigned`". If `data_class` is set to one of these values, input data to the operator is extended in the appropriate way (sign-extended or zero-extended, respectively).

For example, assume that you want to add a 3-bit data value to a 10-bit data value. A 10-bit adder module can accomplish this operation if the 3-bit data is resized. When `data_class` is "`signed`", the 3-bit data is sign-extended to 10 bits. When `data_class` is "`unsigned`", the 3-bit data is zero-extended to 10 bits.

If no `data_class` is declared, the width of the operator's input data must match the width of the module pins exactly.

4.2.3.1.2 permutable_inputs Attribute

The `permutable_inputs` attribute declares a pair of input pins that can be swapped without altering the function of an operator. For example, the two input values to an addition operator can be swapped. Information from the `permutable_inputs` attribute is used during arithmetic optimization.

For a synthetic operator that has a `permutable_inputs` attribute, you must define a separate binding for each permutation. See "Defining Bindings" on page 56 for details.

4.2.3.1.3 Other Attributes

Other operator attributes that aid in the verification of bindings can be included in an `operator` group.

4.2.3.2 Groups within Operators

Every operator group must contain one or more pin groups to specify the names and directions of the operator's ports.

4.2.3.2.1 pin Groups

Operator pins are defined by `pin` groups. There is one such group per pin. Each one contains a single attribute, `direction`, indicating whether the pin is `input` or `output` (inout pins are not allowed in operators). For example:

```
pin(A) {
direction : input;
}
```

4.2.4 Defining Modules

You define synthetic modules by including `module` groups in your synthetic library. Within a `module` group, you declare the pins of the module, bindings to operators, and, optionally, some or all implementations of the module.

The following example shows the module declaration for `SL_addov`, which figures prominently in the tutorial exercises of "Tutorial" on page 21. It contains a `design_library` attribute statement, a `parameter` group, and several pin groups. Module declarations often contain `binding` and `implementation` groups as well; these are discussed separately, in the sections "Defining Bindings" on page 56 and "Declaring Implementations" on page 60.

```
module (SL_addov) {
    design_library : "SL";
    parameter (width) {
       hdl_parameter : TRUE;
    }
    pin (A) {
       direction : input;
       bit_width : "width";
    }
    pin (B) {
       direction : input;
       bit_width : "width";
    }
    pin (CI) {
       direction : input;
       bit_width : "1";
    }
    pin (SUM) {
       direction : output;
       bit_width : "width";
    }
    pin (OV) {
       direction : output;
       bit_width : "1";
    }
    pin (CO) {
       direction : output;
       bit_width : "1";
    }
```

```
    /* binding and implementation groups go here */
}
```

4.2.4.1 Module Attributes

Module groups must include a `design_library` attribute, described below; other, optional attributes are described later in this book.

4.2.4.1.1 design_library Attribute

The `design_library` attribute identifies the design library that contains the module's implementations. You specify the library's logical name, rather than its UNIX path name. For example:

```
design_library : "SL";
```

4.2.4.1.2 Properties Attribute

Currently, we support a few special properties on modules that gives more information to DC. For example:.

```
properties : "boolean:sequential:true string:posedge_clk:clk_s
   string:negedge_clk:clk_d";
```

The properties tells DC that this module is a sequential component; pin "clk_s" is a positive edge clock; pin "clk_d" is a negative edge clock.

4.2.4.1.3 Other Attributes

Optional attributes, which can help you verify the consistency of your synthetic library description, are discussed in "Compiling and Verifying Synthetic Libraries" on page 73.

4.2.4.2 Groups within Modules

A `module` group can contain one or more `parameter` groups, and must contain one or more `pin` groups.

4.2.4.2.1 parameter Groups

Module parameters are declared in `parameter` group statements. You can specify the value of a parameter by including a `formula` attribute statement in the `parameter` group. For example:

```
parameter (X) {
    ...
}
parameter (Y) {
    ...
}
parameter (A) {
    formula : "X + Y" ;
}
```

"Basic Syntax" on page 50 gives details on constructing a `formula` attribute statement.

For a module parameter that corresponds to a VHDL *generic* or a Verilog parameter, you must set the parameter's `hdl_parameter` attribute to `TRUE`. This attribute tells the synthesis tools that the value of the parameter will be provided by the HDL code that instantiates or infers the synthetic module. For example:

```
parameter (width) {
hdl_parameter : TRUE;
}
```

The value of a parameter can be defined by the context of a module if the module instantiation contains a parameter specification (`generic map` in VHDL). When a module is instantiated in an HDL in this way, the synthetic library parameter is set equal to the HDL parameter. In Setting a Parameter by Instantiation (Verilog) and Setting a Parameter by Instantiation (VHDL), the parameter `n` (the first parameter) of a synthetic module is set to `5` by instantiation.

4.2.4.2.2 Setting a Parameter by Instantiation (Verilog)

```
DW01_ADD #(5) ADD1();
```

4.2.4.2.3 Setting a Parameter by Instantiation (VHDL)

```
ADD1: DW01_ADD generic map (n => 5) port map();
```

Each parameter group must include either a formula attribute or a "`hdl_parameter : TRUE;`" statement. If the parameter has a formula attribute and "`hdl_parameter : TRUE;`", then the parameter value set in the HDL instantiation overrides the value specified by the formula attribute.

Parameter definitions apply only within the containing module definition. Before a parameter can be used, it must be defined within the current module. For example, you can use a parameter name for the value of a pin's `bit_width` attribute, but only if the parameter is defined within the module that contains that pin. Parameters defined in a module are available to externally declared implementations of the module.

4.2.4.2.4 pin Groups

The pin declarations in a module group have two attributes. The `direction` attribute defines the pin direction and must be `input`, `output`, or `inout`. The `bit_width` attribute defines the number of bits to be connected to the module pin. For example:

```
pin (A) {
 direction : input;
 bit_width : "width";
}
```

The value of the `bit_width` attribute can be an integer (in quotes) to indicate that the pin has a fixed width. The attribute value can also be the name of a parameter declared within the scope of the `module` statement.

4.2.4.2.5 Other Groups

A `module` group statement can also contain `binding` group statements, `implementation` group statements, and attributes for verification.

4.2.5 Defining Bindings

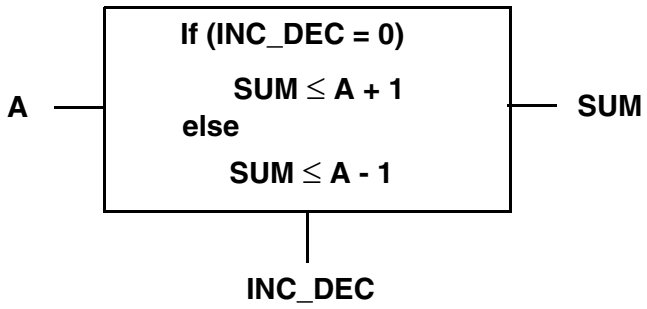
Bindings associate operators with modules. Each binding represents a distinct method for mapping an operator to a module. When the synthesis tools infer a synthetic operator in an end-user's design, they use bindings to find the synthetic module(s) that perform the required function.

You declare a binding with a `binding` group. A `binding` group can appear either inside a module group, or externally. External bindings allow you to bind new operators to the modules in an existing library, even if you do not have access to the source code for that library.

A `binding` group must contain a `bound_operator` attribute and one or more `pin_association` groups. It can also contain one or more `constraint` groups.

The declaration of an incrementer-decrementer module (INCDEC, Figure 4-2) is shown in Binding Declarations.

Figure 4-2 Incrementer-Decrementer



The INCDEC module has two input pins (A, INC_DEC) and one output pin (SUM).

The groups and attributes that appear in the bindings of Binding Declarations are described below.

4.2.5.0.1 Binding Declarations

```
module (INCDEC) {
    ... /* declarations for pins A, SUM, and
       INC_DEC go here */
    ... /* other declarations go here */
    binding(b1) {
       bound_operator : "ADD_UNS_OP";
       pin_association(INC_DEC) { value : "0"; }
       pin_association(A) { oper_pin : A; }
       pin_association(SUM) { oper_pin : Z; }
       constraint(B) { value : "1"; }
    }
    binding(b2) {
       bound_operator : "ADD_UNS_OP";
       pin_association(INC_DEC) { value : "0"; }
       pin_association(A) { oper_pin : B; }
       pin_association(SUM) { oper_pin : Z; }
       constraint(A) { value : "1"; }
    }
    binding(b3) {
       bound_operator : "SUB_TC_OP";
       pin_association(INC_DEC) { value : "1"; }
       pin_association(A) { oper_pin : A; }
       pin_association(SUM) { oper_pin : Z; }
       constraint(B) { value : "01"; }
    }
}
```

4.2.5.1 Binding Attributes

Every binding group must include a bound_operator attribute statement.

4.2.5.1.1 bound_operator Attribute

The bound_operator attribute identifies the name of the synthetic operator being mapped to the current module. In Binding Declarations, the bindings b1 and b2 represent two different mappings from the ADD_UNS_OP (unsigned addition) operator to the INCDEC module. The binding b3 maps the SUB_TC_OP (2's-complement subtraction) operator to the INCDEC module.

4.2.5.2 Groups within Bindings

A `binding` group must contain one or more `pin_association` groups to map operator ports to module ports. It can contain any number of `constraint` groups, which disable the binding for specified conditions.

4.2.5.2.1 pin_association Groups

The `pin_association` group statement defines how the module pins are associated with synthetic operator pins and constant values. In every binding, each connected pin of a module must have a corresponding `pin_association` group with an identical name. If a module pin does not have a `pin_association` group, it is left unconnected.

A `pin_association` group can contain either a `value` attribute or an `oper_pin` attribute, as in

```
pin_association(INC_DEC) { value : "0"; }
pin_association(A) { oper_pin : A; }
```

The `value` attribute defines a constant (binary) value to be connected to the module pin. The `value` attribute is most useful for defining values on a module's function control pins. In binding `b1` of Binding Declarations, the `INC_DEC` pin of the module is connected to a constant value of `0`.

If the bit-width of the specified value is less than the width of the module pin, the value is either sign-extended or zero-extended to fit the width of the pin. The method of extension selected depends on the `data_class` of the operator (see "Defining Operators" on page 52).

The `oper_pin` attribute associates a synthetic operator pin with the module pin of the `pin_association` group. The external signal connected to the operator pin in the end-user's design gets connected to the module pin of the `pin_association` group.

For instance, in Binding Declarations, if the binding `b2` is used, the signal connected to pin `B` of the `ADD_UNS_OP` operator gets connected to pin `A` of the `INCDEC` module. If binding `b1` is used instead, the signal connected to pin `A` of the `ADD_UNS_OP` operator gets connected to pin `A` of the `INCDEC` module. Two separate bindings (`b1` and `b2`) are defined to make the input operands permutable.

If the data width of the operator pin differs from the data width of the module pin, the data value is either sign-extended or zero-extended to fit the width of the module pin. The method of extension selected depends on the `data_class` of the operator (see "Defining Operators" on page 52).

Bindings can be used to associate a synthetic module with a synthetic operator even when there is not a one-to-one correspondence between the pins of the module and the pins of the operator. This advanced feature is discussed in "Optional Pins" on page 67.

4.2.5.2.2 constraint Groups

A `constraint` group specifies conditions, based on the data values present on the bound operator's pins, under which the module implements the functionality of the bound operator. If the conditions are not met, the module cannot perform the function required by the operator, and the binding is not used.

If a `binding` group contains more than one `constraint` group, all the constraints must be met in order for the binding to be used.

The name of the constraint corresponds to the name of the operator pin of the `bound_operator` attribute (rather than the name of a `pin_association` group, which corresponds to a module pin). For example, as in the following fragment from Binding Declarations:

```
binding(b1) {
    bound_operator : "ADD_UNS_OP";
    ...
    constraint(B) { value : "1"; }
}
```

A `constraint` group must contain a single `value` attribute. `value` defines the (binary) data value that must be connected to the corresponding operator port. If the designated value is not connected to the port, the binding is not used. Constraint values are either sign-extended or zero-extended, as determined by the `data_class` of the operator.

In Binding Declarations, the binding `b2` is used only if pin `A` of `ADD_UNS_OP` is connected (in the end-user's netlist) to the constant value `1`:

```
constraint(B) { value : "01"; }
```

The constraint value is `01`, which represents a positive value of `1` in 2's-complement form. Two digits are used in this value because `SUB_TC_OP` is a 2's-complement operator, and a single-digit value cannot be sign-extended to a positive value. You must include sign bits when binding to a signed operator.

4.2.5.3 External Bindings

You can also declare bindings outside the body of the target module. Such bindings are called *external bindings*. The target module can even be in a separate synthetic library from that of the binding.

You define an external binding in the same way you define an ordinary binding, except that you include the name of the target module in the binding declaration. The following example shows how to create an external binding `b3` for module `M1`.

```
binding(b3, M1) {
    /* body of the binding (like ordinary bindings) */
}
```

The associated module and the bound operator do not have to be declared in the same synthetic library file as the binding. Name references are resolved using the libraries defined in the `dc_shell-t` variable `synthetic_library`. The first module or operator encountered with the correct name will be used.

The `report_synlib` command reports on external as well as ordinary bindings.

4.2.6 Declaring Subblocks

A *subblock* is a DesignWare Building Block IP component that has only one implementation and is always instantiated rather than inferred. Subblocks are useful for large parts, such as a microsequencer, an Ethernet controller, or a SCSI controller, since users most often want a particular microsequencer, for example, rather than have a synthesis tool choose one from a number of implementations. Using subblocks also reduces memory usage and can speed synthesis of a design.

You create subblocks, like modules, with an HDL or a `.db` file. You do not declare ports, bindings, or parameters in a subblock declaration.

The declaration syntax within the `.sl` file for a subblock is simple. The following example shows the syntax.

```
subblock (subblock_name) {
    design_library: design_library_name
    implementation(implementation_name) {
    }
}
```

For VHDL designs, subblock_name must match the VHDL entity name, and implementation_name must match the VHDL architecture name. For Verilog designs, the module name must be of the form

```
subblock_name__implementation_name
```

For example, the Verilog module for the foo implementation of the subblock bar would be named bar__foo.

The report_synlib command can be used to list the subblocks in a .sl file.

4.2.6.1 Hierarchy in Subblocks

Subblocks can have a hierarchical structure. The following design objects may be used within a subblock:

- Another DesignWare Building Block IP component (subblocks, modules, operators)
- A cell from the gtech library
- A technology-specific library cell

When compiling a subblock, a different search path and link library are used than when compiling a synthetic library module. Therefore, if a subblock hierarchy requires linking to other .db or .sldb files, you must tell end users explicitly which directories to include in their search_path variable and which .sldb files to include in their synthetic_library and link_library variables. To declare a subblock that has hierarchy, it is convenient to declare the levels of hierarchy in the same .sl file, so that all the sub-designs are found automatically.

4.2.7 Declaring Implementations

An *implementation declaration* associates a synthetic module with its implementations in a design library. You declare implementations with implementation groups. An implementation group can either be included in the scope of the module group, or it can be outside the module group (*external implementation*).

The names of implementation groups are declared in the same way as any other type of group - as a single word in parentheses. These names must match those of the designs they refer to, according to the rules given in "Creating DesignWare Implementations" on page 39.

In the following example, the implementations named rpl and cla are declared as possible implementations of the module SL_addov.

```
module (SL_addov) {
    /* other declarations go here */
    implementation (rpl) {
    }
    implementation (cla) {
    }
}
```

4.2.7.1 Implementation Attributes

An implementation group statement can contain several different attributes, most of which are optional. The technology attribute is required for technology-specific parts.

4.2.7.1.1 technology Attribute

The technology attribute defines the target technology of the implementation. For an implementation to be selected by the synthesis tools, the value of the technology attribute must match the current value of the

target_technology variable. Set this attribute on your implementation if and only if the implementation instantiates cells from a particular technology library.

For example, the proprietary implementation of SL_addov (see "Tutorial" on page 21) contains a technology attribute statement:

```
implementation (proprietary) {
    technology : my_tech.db;
}
```

The example below enables the proprietary implementation of SL_addov to access the library my_tech.db.

```
target_library = { my_tech.db my_tech_pads.db }
```

Only the implementations with no technology attribute or with a technology attribute of my_tech.db are used.

Do not change the order of the arguments in the target_library variable; this causes a new synthetic cache model to be built.

4.2.7.1.2 design_library Attribute (Optional)

The design_library attribute identifies the design library that contains the design for the implementation. If absent, the design library of the synthetic module is used. A design_library attribute on an implementation takes precedence over a design_library attribute on the module.

4.2.7.1.3 Other Attributes

Attributes for controlling implementation selection are discussed in "Creating Synthetic Libraries: Advanced" on page 65.

Attributes that can help you in verifying the consistency of your synthetic library description are discussed in "Compiling and Verifying Synthetic Libraries" on page 73.

4.2.7.2 area_estimate Parameter Group (Optional)

When performing area_only optimization, the synthesis tools do not need a full pre-optimization to obtain models. You can apply the optional area_estimate attribute to your implementation when you have no timing constraints on your design. The formula of this parameter is used to model the cell's area, which saves CPU time by avoiding pre-optimization.

To make the area_estimate parameter technology-independent, use the size function. size returns the area of an implementation. You use the size function to characterize the area of your implementation in terms of the area of smaller implementations. The synthesis tools optimize these smaller implementations in the target technology to model their area. The area_estimate parameter uses the estimates to model the entire cell's area.

The following example shows (in bold) the area estimator for a ripple-carry adder. Notice that the area estimate relies on the value of other parameters (the pin width n) of the current module as well as on the size of a smaller synthetic library implementation (DW01_ADD_ABC) in the current target technology.

```
module(MY_ADD) {
    parameter(n) {
        formula : "max(width('a'), width('b'), width('sum'))"
        hdl_parameter : TRUE
    }
    pin(a) {
        direction : input;
        bit_width : "n";
    }
    pin(b) {
        direction : input;
        bit_width : "n";
    }
    pin(sum) {
        direction : output;
        bit_width : "n";
    }
    implementation(ripple) {
        parameter (area_estimate) {
        formula : " n * size('DW01_ADD_ABC') "
        }
    }
}
```

4.2.7.3 External Implementation Declarations

An external implementation declaration is an `implementation` group that occurs outside the associated `module` group. External declarations allow you to declare implementations for the modules in an existing library, even if you do not have access to the source code for that library.

The name of an external `implementation` group has two fields. The first field is the implementation name. The second field is the name of the module associated with the implementation. In the following example, `proprietary` is declared as an implementation of the module `SL_addov`.

```
implementation (proprietary,SL_addov) {
    technology : class.db;
    parameter (legal) {
        formula : "width == 8";
    }
}
```

4.3 Mapping HDL Operators to Your Components

You map an HDL operator to a synthetic operator by including the `map_to_operator` pragma in the definition of the HDL operator. You can see examples of the use of this pragma in the `std_logic_arith` package. This package is located under the Synopsys root directory in the file `packages/IEEE/src/std_logic_arith.vhd`.

The names of the formal input parameters of the HDL operator must match the input port names of the synthetic operator. The formal output parameters of the HDL operator must have the same names as the output ports of the synthetic operator. You can use the `report_synlib` command to find out the port names of operators in a synthetic library.

The restriction of matching parameters to ports is somewhat relaxed when you use the optional pin mechanism described in "Creating Synthetic Libraries: Advanced" on page 65.

You associate a function return value with the output port of a synthetic operator by using a second pragma, `return_port_name`, as shown in the following VHDL and Verilog "map_to_operator" examples.

4.3.0.0.1 VHDL map_to_operator

```
function my_unsigned_add(A, B : bit_vector) return bit_vector is
    -- synopsys map_to_operator ADD_UNS_OP
    -- synopsys return_port_name Z
begin
--
-- Code used for simulation goes here
--
end my_signed_add;
```

4.3.0.0.2 Verilog map_to_operator

```
function [15:0] my_signed_mult;
    // synopsys map_to_operator MULT_TC_OP
    // synopsys return_port_name Z
input [7:0] A, B;
/*
 * Code used for simulation goes here
 */
endfunction
```

When you use the `map_to_operator` pragma, the HDL Compiler parses and ignores the body of the function. The function body is used only for simulation.

Make sure that the functionality of the simulation code matches the functionality of the synthetic operator, especially with boundary conditions and side effects. Failure to match the simulation code with the operator function causes mismatches between pre- and post-synthesis simulation results.

The `map_to_operator` pragma also works with VHDL procedures and Verilog tasks.

A comprehensive list of Synopsys operators is listed in "Standard Synthetic Operators" on page 85.

5 Creating Synthetic Libraries: Advanced

Certain advanced features of the synthetic library description language—*legality* and *priority*—give you a measure of control over the implementation selection process. For example, you can tell the synthesis tools that a particular implementation of a parametrized module should not be used for a given range of one of its parameters.

Another advanced feature allows you to create synthetic operators with optional pins. Such operators can be bound to modules that lack the pins marked as optional.

This chapter describes

5.1 Legality and Priority

This section introduces two capabilities that give you some control over implementation selection, *legality* and *priority*:

- You can specify parameter ranges for which a particular implementation is *legal*.
- You can assign *priorities* to the various implementations of a given module.

You specify legality and priority as functions of module parameters. For example, you can make a particular adder implementation low priority when the operands are less than 9 bits wide, high priority for 9–16-bit applications, and illegal for operands wider than 16 bits.

For any particular set of parameter values, the resource-sharing and implementation-selection algorithms consider only legal implementations, and, among legal implementations, only those that have the highest priority.

In general, there will be a set of legal implementations that tie for the highest priority. The optimization algorithms choose the best implementation from this set.

Design Compiler users can temporarily change priorities or disable priority checking. They cannot disable legality checking.

5.1.1 Implementation Legality

The legality capability can be used for two purposes:

- To specify parameter ranges for which a particular implementation is not available. For example, certain hard macros for addition are not available in bit widths greater than 16.
- To specify parameter ranges for which a particular implementation should not be considered because the implementation is grossly inferior to others under the specified conditions. In this case, the goal is to speed up the optimization process by providing hints on what are the best implementations for various bit ranges.

You define legality for an implementation by including a `parameter` group named `legal` in the implementation declaration. This parameter group contains a single formula specifying the conditions under which the implementation can be used (as in the following example).

```
implementation (proprietary,SL_addov) {
    technology : class.db;
    parameter (legal) {
        formula : "width == 8";
    }
}
```

If the `width` parameter of an instance of the module `SL_addov` is not equal to 8, the formula in the above example evaluates to 0, and the `proprietary` implementation cannot be used for that module instance.

Design Compiler users cannot turn off legality testing. Illegal implementations will never be considered by the optimizer.

There should be at least one legal implementation of a module for any valid set of parameter values. The legal implementations may be different for different values, but there should always be at least one. The tool exits and reports an error if no legal implementation exists for a module instance, whether the module was inferred or explicitly instantiated by the user.

5.1.2 Implementation Priority

You use the priority capability to express your preferences among available implementations of a given module. Priorities help the optimizer to arrive at good solutions faster by restricting the search space.

An implementation priority is an integer function of module parameters. The priority formula must evaluate to an integer between 0 and 10 for all valid combinations of parameter values. An implementation with no priority assigned to it will have priority 5. If, for given parameter values, the priority formula yields a value less than 0 or greater than 10, the priority defaults to 5 and the tool issues a warning.

An implementation is visible to the optimizer only if all higher priority implementations of the same module are illegal.

You define the priority of an implementation by including a `parameter` group named `priority` in the implementation declaration. This parameter group contains a formula that evaluates to the priority (as in the following example).

```
implementation (rpl, ADD) {
    library: "DW01";
    parameter(priority) {
        formula: " n < 4 ? 8 : 0" ;    /* good until 4 */
    }
}
```

The priority formula in the above example uses the conditional operator (?). (See "Creating Synthetic Libraries: Basic" on page 47 for details about constructing formulas.) For instances of `ADD` whose bit-width parameter `n` is less than 4, the implementation `rpl` has priority 8. For instances of `ADD` with `n >= 4`, `rpl` has priority 0.

Note Priorities distinguish only among implementations of the same module. The priorities of implementations of different modules are unrelated and hence cannot be used to communicate preferences.

Sometimes it is best not to preset priority values in your implementation definitions. If your users have more than one supplier of implementations for the same modules, your priority scheme and that of other suppliers may be in conflict. For example, you may think of 6 or 7 as "high-priority," while another supplier regards 9 or 10 as "high-priority." Your implementations may never get selected, regardless of their size and speed.

Users often have access to libraries from many sources; in such cases, they should establish their own system of priorities, one that takes into account all the implementations available to them.

End users can temporarily override priority values or disable priority testing. The `dc_shell-t` command, `set_impl_priority`, lets users override priorities specified in the synthetic library without modifying the library itself. In this way, users can compare results as they vary the priorities. See the man page for `set_impl_priority` for full details.

End users can disable priority testing by setting a new `dc_shell-t` variable, `hlo_ignore_priorities`, to `"true"`.

5.2 Optional Pins

Synthetic libraries often include operators that are very similar to each other, but would seem to require separate declarations because they have different numbers of pins. For example, the operators `ADD_UNS_OP` and `ADD_UNS_CI_OP` differ only by the carry-in pin, `CI`, and might therefore require two operator declarations and two bindings for every module that uses them. Some operators with similar functions differ by more than one pin; the number of bindings required would grow exponentially with the number of extra pins.

A more flexible mechanism for binding operators to modules reduces the number of operators and bindings that need to be declared.

5.2.1 Creating Synthetic Operators with Optional Pins

In general, the process of mapping an HDL subprogram to a synthetic module occurs in two steps:

1. The formal parameters of the HDL subprogram are mapped to the input and outputs pins of the synthetic operator.

2. The synthetic operator pins are mapped to synthetic module pins by means of bindings.

The following features of bindings give you a measure of control over these mappings, making it possible for you to associate HDL subprograms, synthetic operators, and synthetic modules with each other, even when their I/O structures differ:

- The `unbound_oper_pin` group—Specifies a value to be assigned to an operator input pin when the HDL subprogram does not associate a formal input parameter with the pin. `unbound_oper_pin` groups must have constants as values.

  Operator output pins that are not associated with formal parameters of the HDL subprogram are simply left unconnected.

  This mechanism makes it possible for you to map an HDL subprogram to a synthetic operator with more pins than the subprogram has parameters.

- The constraint value "`unbound`"—Invalidates a binding in those cases where an HDL subprogram uses any of the synthetic operator pins that are missing from the synthetic module.

  `constraint` groups apply only to input pins. For an operator output pin that is not in the module, you provide no `pin_association` in the binding. The binding is then automatically invalidated for HDL subprograms that use the pin.

  This mechanism makes it possible for you to bind a synthetic operator to a synthetic module that has fewer pins than the operator.

The `unbound_oper_pin` group applies in Step 1; the `constraint` group applies in Step 2. The bindings in Binding with Optional Pins and Another Binding with Optional Pins illustrate the optional-pin capabilities.

5.2.1.0.1 Binding with Optional Pins

```
VHDL:

procedure unsigned_add_ci_co(A, B: in UNSIGNED;
SUM: out unsigned; CO: out bit);
-- pragma map_to_operator ADD_UNS_CI_CO_OP

.sl:

operator(ADD_UNS_CI_CO_OP) {
    pin(OA) {
       direction : input;
    }
    pin(OB) {
       direction : input;
    }
    pin(OCI) {
       direction : input;
    }
    pin(OSUM) {
       direction : output;
    }
    pin(OCO) {
       direction : output;
    }
}

module(add_ci_co) {
```

```
    pin(A) {
       direction : input;
       bit_width : "n";
    }
    pin(B) {
       direction : input;
       bit_width : "n";
    }
    pin(CI) {
       direction : input;
       bit_width : "1";
    }
    pin(SUM) {
       direction : output;
       bit_width : "n";
    }
    pin(CO) {
       direction : output;
       bit_width : "1";
    }
    binding(b1) {
       bound_operator : "ADD_UNS_CI_CO_OP";
       pin_association(A) { oper_pin : OA; }
       pin_association(B) { oper_pin : OB; }
       pin_association(CI) { oper_pin : OCI; }
       pin_association(CO) { oper_pin : OCO; }
       pin_association(SUM) { oper_pin : OSUM; }
       unbound_oper_pin(OCI) { value : "0" }
    }
...
...
}
```

The operator input pin `OCI` is not present in the HDL subprogram (`OCI` is *unbound*). The synthesis tools can use binding `b1` by connecting the module input `CI` (associated with `OCI`) to 0. By default, if any operator output pin is unbound, the binding can be used, with the associated module output left unconnected.

5.2.1.0.2 Another Binding with Optional Pins

VHDL:

```
procedure basic_add(A, B: in UNSIGNED; SUM: out unsigned);
-- pragma map_to_operator ADD_UNS_CI_GREY_CO_OP
```

.sl:

```
operator(ADD_UNS_CI_GREY_CO_OP) {
    pin(OA) {
       direction : input;
    }
    pin(OB) {
       direction : input;
    }
    pin(OCI) {
       direction : input;
    }
    pin(OGREY) {
       direction : input;
```

```
        }
        pin(OSUM) {
            direction : output;
        }
        pin(OCO) {
            direction : output;
        }
    }

    module(stripped_down_add) {
        pin(A) {
            direction : input;
            bit_width : "n";
        }
        pin(B) {
            direction : input;
            bit_width : "n";
        }
        pin(SUM) {
            direction : output;
            bit_width : "n";
        }
        binding(b2) {
            bound_operator : "ADD_UNS_CI_GREY_CO_OP";
            pin_association(A) { oper_pin : OA; }
            pin_association(B) { oper_pin : OB; }
            pin_association(SUM) { oper_pin : OSUM; }
            constraint(OCI) { value : "0";}
            unbound_oper_pin(OCI) { value : "0"; }
            constraint(OGREY) { value : "unbound";}
        }
    ...
    ...

    }
```

Binding `b2` maps an operator to a module with fewer pins. The module has no pins associated with the operator's `OCI` and `OGREY` inputs and `OCO` output:

- If `OCI` appeared as a parameter in the HDL subprogram, the `OCI` constraint would ensure that binding `b2` is used only when `OCI` is connected to 0. Since `OCI` is omitted from the HDL parameter list, the `unbound_oper_pin` group assigns it the value 0; this ensures that the `OCI` constraint is met.
- The `OGREY` constraint would invalidate the binding if `OGREY` were bound to an HDL parameter.
- If the `OCO` pin were bound to an HDL parameter, then binding `b2` would be invalid, since there is no `pin_association` for `OCO`. Since `OCO` does not appear in the HDL, the binding can be used.

5.2.2 Rules and Restrictions

These are some basic rules for using optional pins in bindings:

- All OPERATOR input pins must appear in the binding (to implement the operator's semantics), otherwise the binding is invalid. The inputs can appear as either in the pin-association, unbound-oper-pin, or constraint.

- All SUBPROGRAM output pins must map to operator pins which must appear in the binding (so that they can be driven) else the binding is invalid. Note: there may be operator outputs that are mentioned in the binding but do not appear as subprogram outputs.
- If an HDL subprogram INPUT pin is missing, `unbound-oper-pin` gives the corresponding operator pin a value (so it can be mapped to a module pin). If there is no `unbound-oper-pin` for the missing pin, then the binding is not valid (unless there is a constraint for the input to be unbound). This is how optional input pins are created.

5.2.3 Mapping HDL Subprograms to Synthetic Operators with Optional Pins

The `map_to_operator` pragma can be used to map procedures to operators even when the parameter list of the procedure does not exactly match the pin list of the operator. For example, given the synthetic library code of Binding with Optional Pins, the following VHDL code defines a valid procedure-to-operator mapping:

```
procedure basic_add(A, B: in UNSIGNED; SUM: out UNSIGNED);
-- pragma map_to_operator ADD_UNS_CI_GREY_CO_OP
```

6 Compiling and Verifying Synthetic Libraries

After you have created your synthetic library code (`.sl` file), you must compile it into a working synthetic library (`.sldb` file).

After compilation, you are ready to verify your synthetic library. For each synthetic module you need to make sure that all implementations of the module perform the correct function. Simulation is the best means of performing this verification.

You must also verify that the bindings you have established between synthetic operators and synthetic modules contain the correct connectivity information and result in proper use of each module.

This chapter covers the following tasks:

6.1 Compiling Synthetic Library Code

You compile synthetic library code in one of two ways, depending on whether you want to create a new synthetic library, or augment an existing one:

- To create a new synthetic library – Your `.sl` file should contain a single library group. You use the `read_lib` and `write_lib` commands to generate a `.sldb` file on disk. The script in the following example illustrates this process.

```
read_lib my_synlib.sl
write_lib my_synlib.sldb
```

- To augment an existing library – You can also add individual synthetic operators, synthetic modules, bindings, and implementations to an existing synthetic library. Define `operator` or `module` groups, or external (two-operand) `binding` or `implementation` groups in your `.sl` file; do not place these groups within a library group. Use the `update_lib` command in `dc_shell-t` as shown in the following example.

```
read synlib1.sldb
update_lib synlib1.sldb my_addition.sl
write_lib synlib1.sldb
```

Note Do not use `update_lib` on the synthetic library `standard.sldb` because parts in `standard.sldb` are licensed differently from user parts.

6.1.1 Verifying Synthetic Library Sets

DesignWare Developer has the ability to validate the simultaneous use of synthetic libraries (`.sldb` files) used as a set. The means to check a set of synthetic libraries is through the `dc_shell-t` command, `check_synlib`.

Users can request error checking over sets of synthetic libraries using the `check_synlib` command. The command usage is:

```
check_synlib
```

There are no command line options. The command checks synthetic libraries found in the paths normally used to search synthetic library paths. The `standard.sldb` library is automatically checked.

6.2 Verifying Implementations and Bindings

In previous releases of Synopsys tools, HDL functions (for example, `+`, `-`, `*`, and `<`) were built-in. For example, from an `a+b` operation in HDL, Synopsys tools automatically generated an adder to perform addition in a manner consistent with simulation.

With the advent of user-defined synthetic libraries, you can now dictate how built-in functions are implemented. However, this added flexibility allows you to make mistakes. You can conceivably define adders that do not add. And if you define a binding incorrectly, the corresponding operator and module might be connected incorrectly.

For these reasons, verification of your implementations and bindings is a crucial step in the process of defining synthetic libraries.

7 Distributing DesignWare Building Block IP

The material in this chapter is addressed to developers of DesignWare Building Block IP who want to distribute the components they develop to outside customers. If your DesignWare Building Block IP components are used only for internal purposes and you do not wish to encrypt them, you need not read this chapter. Synopsys no longer provides a method for licensing your components.

This chapter describes a production environment and the procedure for encrypting and shipping DesignWare Building Block IP components.

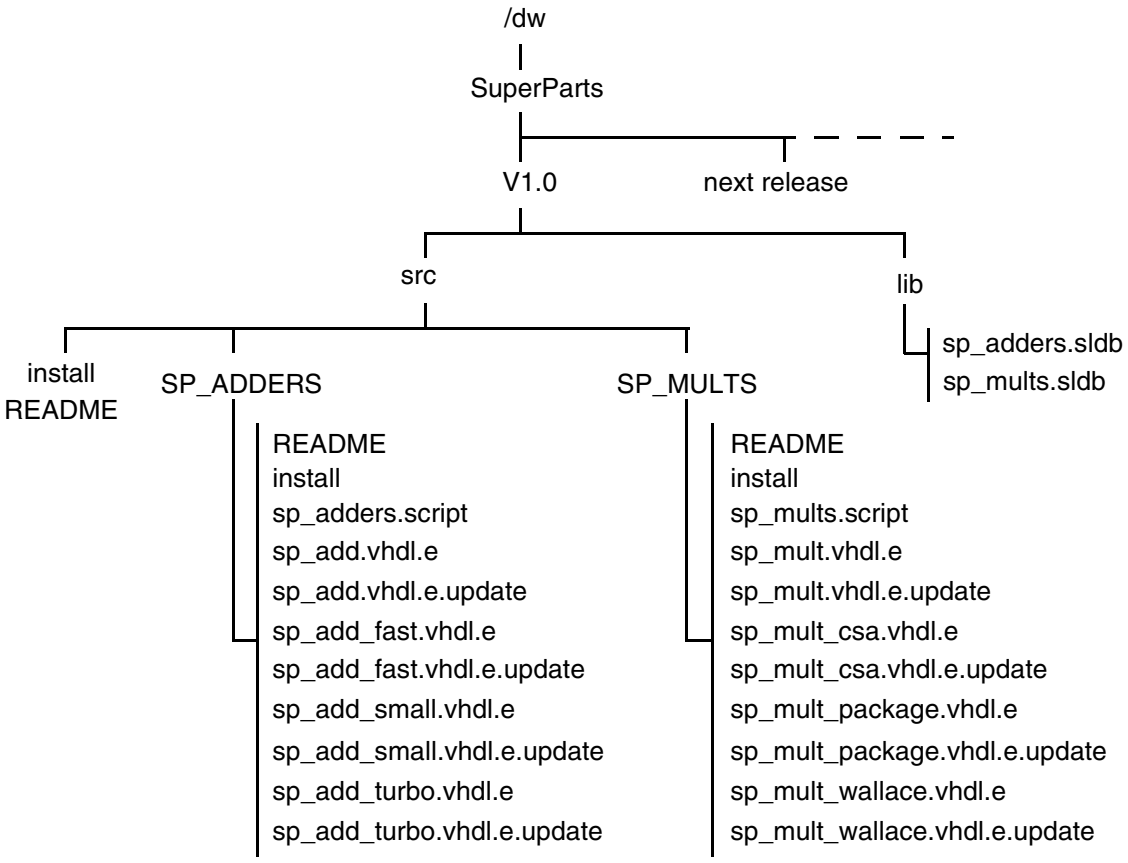
You may want to protect your intellectual property by not shipping HDL source code for your DesignWare Building Block IP components. The encryption capability allows you to ship encrypted HDL source files—and, therefore, platform-independent libraries—while protecting the proprietary contents of your design. The encrypted source files are processed by Synopsys tools in the same way that unencrypted HDL source files are processed.

Creating production DesignWare Building Block IP components involves the following tasks:

7.1 Creating the Recommended Directory Structure

Figure 7-1 shows the recommended directory structure for production of your Building Block IP and their subsequent installation at your customer sites. To illustrate the basic concepts, we use a fictitious components vendor named SuperParts, Inc., throughout the chapter. This vendor has two Building Block IP families, `SP_ADDERS` and `SP_MULTS`.

Figure 7-1 Directory Structure for Production



The `SP_ADDERS` library contains three implementations of a basic adder called `add_fast`, `add_turbo`, `add_small`. The entity declaration for this component is in the encrypted VHDL file `sp_add.vhdl.e`. The architecture for the three implementations are in encrypted VHDL files `sp_add_fast.vhdl.e`, `sp_add_turbo.vhdl.e`, and `sp_add_small.vhdl.e`.

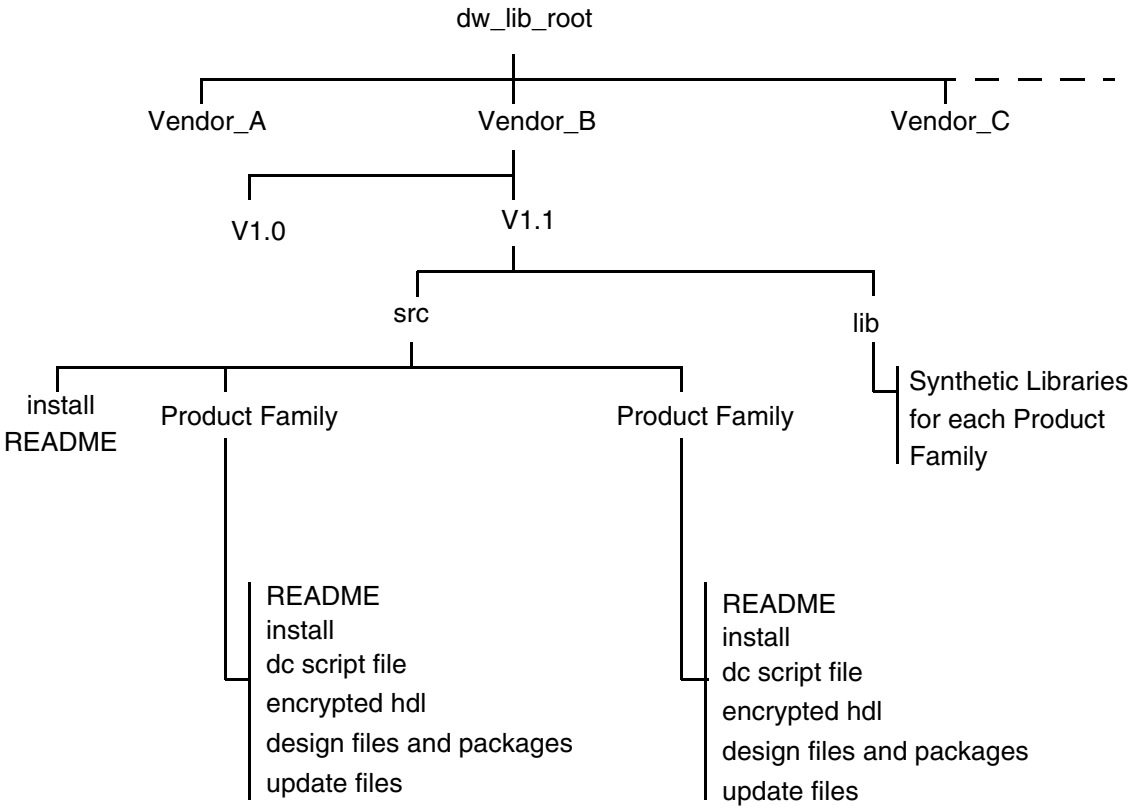
Similarly, the `SP_MULTS` library contains two implementations of a multiplier. The entity declaration for this component is in file `sp_mults.vhdl.e`. The architectures for the two implementations are in encrypted VHDL files `sp_mults_wallace.vhdl.e` and `sp_mults_csa.vhdl.e`. The file `sp_mult_package.vhdl.e` contains common declarations and subprograms used by the multiplier family.

Along with each encrypted source file is a `.update` file. The update files allow your customers to analyze the corresponding `.vhdl` files even if only Verilog HDL Compiler license(s) are available. You create update files with the `analyze -create_update` command (see below).

Each family requires an installation script, a `dc_shell-t` script file, a README file, and an associated synthetic library file (`.sldb`), which resides in the `lib` directory.

The directory structure illustrated in Figure 7-2 enables your customer to maintain DesignWare Building Block IP components from different vendors. It also helps them maintain different versions of several product families from a single vendor.

Figure 7-2 Directory Structure for DesignWare User



`dw_lib_root` indicates a directory in which all Building Block IP are installed. It could be any directory. In Figure 7-3, `dw_lib_root` is mapped to `/dw`. Once your customer has installed your DesignWare Building Block IP, you must advise your customer to use the `dc_shell-t` command `analyze` (described later in this chapter) to populate the design libraries. These design libraries are created in a parallel directory structure relative to `src`.

Thus for each DesignWare family under

```
dw_lib_root/vendor/version/src/family
```

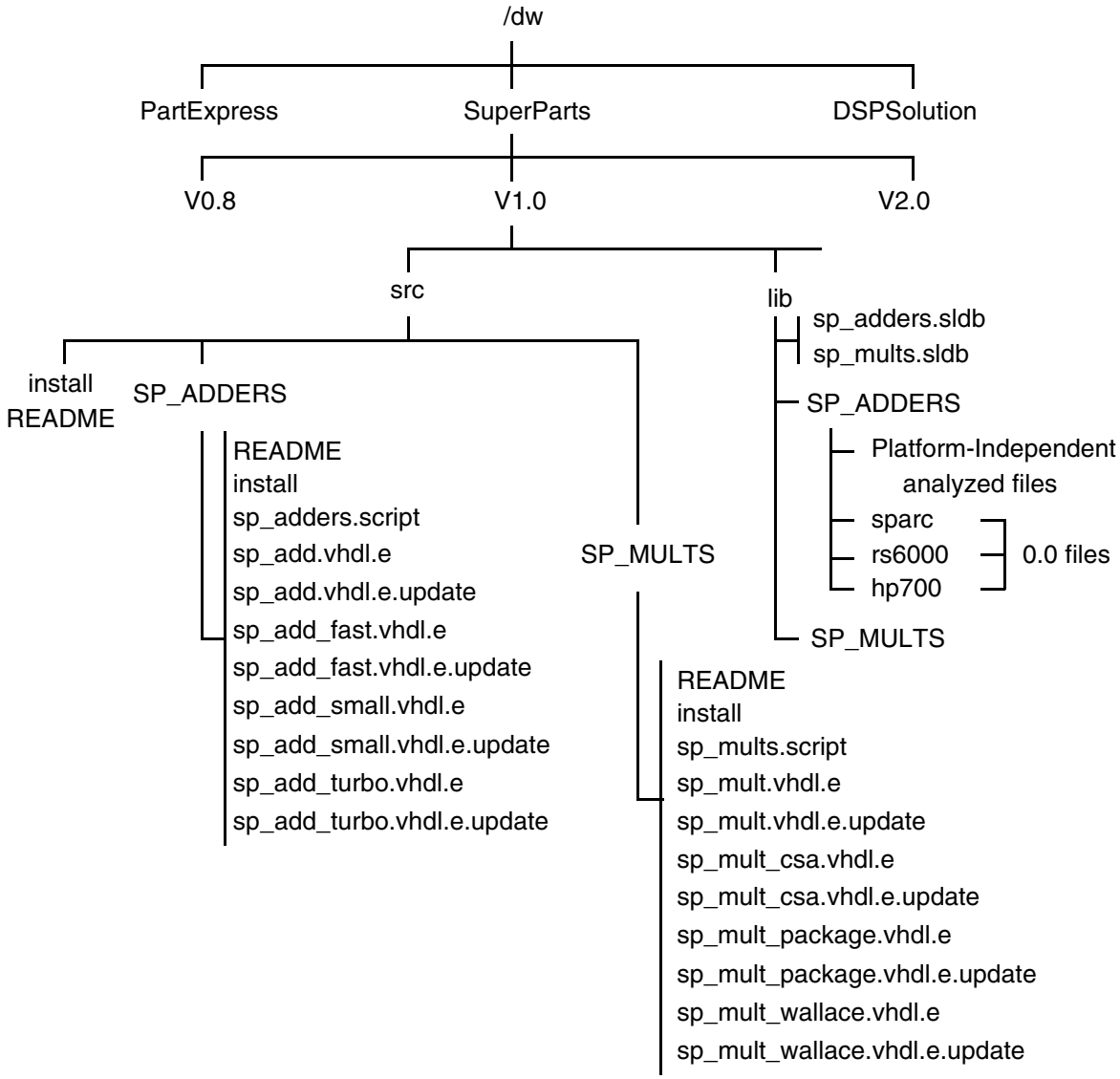
there is a corresponding directory at

```
dw_lib_root/vendor/version/lib/family
```

The analyzed files your customer will generate are platform-independent.

For example, a SuperParts customer site would have the directory structure shown in Figure 7-3 after installing and generating design libraries for `SP_ADDERS` and `SP_MULTS`. The diagram assumes that the customer is using a heterogeneous network with three different machine architectures.

Figure 7-3 Directory Contents at Customer Site after Installation



7.2 Creating Production DesignWare Building Block IP Component Libraries

Synopsys recommends that you follow these steps for the proper release and shipment of DesignWare Building Block IP to your customers. The following six steps are detailed in this section.

1. "Step 1: Create the directory structure for release" on page 79
2. "Step 2: Compile the synthetic library code" on page 79
3. "Step 3: Encrypt the HDL source" on page 79
4. "Step 4: Create the update files" on page 80
5. "Step 5: Create dc_shell-t script" on page 81
6. "Step 6: Create installation instructions and script" on page 82

7.2.1 Step 1: Create the directory structure for release

DesignWare Building Block IP are usually organized into product categories. The directory structure on the distribution tape for each Building Block IP category should look like:

```
vendor/version/src/family
vendor/version/lib
```

You must create a directory for each family in the `src` directory. You should create only one `lib` directory. See Figure 7-1 on page 76.

The remaining steps in this process show you how to populate these newly created directories.

7.2.2 Step 2: Compile the synthetic library code

Procedures for defining synthetic libraries are described in "Creating Synthetic Libraries: Basic" on page 47 and "Creating Synthetic Libraries: Advanced" on page 65. Compilation is covered in "Compiling and Verifying Synthetic Libraries" on page 73. You must supply your customers with a compiled version of the synthetic library file that you have created for each of your Building Block IP categories. Put these files in the directory `vendor/version/lib`.

Create the synthetic library database file (`.sldb`) by reading the synthetic library source file in `dc_shell-t`. You can create the `.sldb` files in another directory and then copy them to this directory. For the SuperParts example (Figure 7-1 on page 76), this directory would be `SuperParts/V1.0/lib`. The synthetic library database files for the `SP_ADDERS` and `SP_MULTS` DesignWare Building Block IP component categories would be `sp_adders.sldb` and `sp_mults.sldb`, respectively.

`sp_adders.sldb` is created by `dc_shell-t` as follows:

```
dc_shell-t> read_lib sp_address.sl
dc_shell-t> write_lib sp_adders.sldb
```

`sp_mults.sldb` is created by `dc_shell-t` as follows:

```
dc_shell-t> read_lib sp_mults.sl
dc_shell-t> write_lib sp_mults.sldb
```

7.2.3 Step 3: Encrypt the HDL source

To encrypt the HDL source, run the Synopsys encryptor, `synenc`, in each subdirectory corresponding to each family of components. The Synopsys encryptor converts the HDL source files to encrypted versions in the current directory. You invoke the encryptor from the UNIX shell as follows:

```
synenc [-r synopsys_root] file1 [file2...fileN]
```

The `-r` option specifies the Synopsys root directory and is optional if the `SYNOPSYS` environment variable is defined. Refer to the `synenc` man page for full details.

Also note the following about the `synenc` command:

- It requires either a DesignWare Developer license or a DesignWare license. The command errors out when the required license is not available.

  License queuing functionality allows you to wait for a license to become available if all licenses are in use. To enable this functionality, set the SNPSLMD_QUEUE environment variable to true before you start the `synenc` command.

When you invoke synenc, it displays the following message:

```
Information: License queuing is enabled. (SYNENC-12)
```

The SNPS_MAX_WAITTIME variable specifies the maximum wait time in seconds for the license. The default wait time is eight hours.

While the original HDL source files can be organized arbitrarily, Synopsys recommends that the HDL source files for each family be put in a separate directory. For example, SuperParts, Inc., could choose to organize the original HDL source files under the root directory dwsrc.

To create the encrypted source in this case, SuperParts, Inc., would execute the following commands in the UNIX shell:

```
% cd SuperParts/V1.0/src
% cd SP_ADDERS
% synenc -r /usr/cad/synopsys /dwsrc/SP_ADDERS/*.vhdl
% cd ../SP_MULTS
% synenc -r /usr/cad/synopsys /dwsrc/SP_MULTS/*.vhdl
```

These commands would create encrypted files in their respective directories. For example, the directory SuperParts/V1.0/src/SP_ADDERS would contain sp_add.vhdl.e, sp_add_fast.vhdl.e, sp_add_small.vhdl.e, and sp_add_turbo.vhdl.e. Note that /usr/cad/synopsys corresponds to the Synopsys root directory.

7.2.4 Step 4: Create the update files

Each encrypted source file should be shipped with a corresponding .update file. These files allow a user to install components whose source is written in either VHDL or Verilog, regardless of the HDL license type the customer has.

You create update files by using the analyze command with the -create_update option. The command writes a file with a .update filename extension, and puts it in the same directory as the HDL source file. By distributing .update files along with the HDL source files, you make it possible for any customer to analyze VHDL source files by using the analyze -update command, regardless of what kind of HDL license the customer has.

The analyze command creates Synopsys intermediate files (.syn, .mra, and so on) as well as the .update files. You do not need these intermediate files, but you do need a place to put them.

Create directories to hold them temporarily:

```
mkdir ~/analyze_temp/sp_adders_temp
mkdir ~/analyze_temp/sp_mults_temp
```

You can then create the update files in dc_shell-t as follows:

```
define_design_lib SP_ADDERS_TEMP -path /temp/sp_adders_temp
analyze -f vhdl [list sp_add.vhdl.e] -w SP_ADDERS_TEMP -create_update
analyze -f vhdl [list sp_add_fast.vhdl.e sp_add_small.vhdl.e] \
  -w SP_ADDERS_TEMP -create_update
analyze -f vhdl [list sp_add_turbo.vhdl.e] -w SP_ADDERS_TEMP -create_update

define_design_lib SP_MULTS_TEMP -path /temp/sp_mults_temp
analyze -f vhdl [list sp_mult_package.vhdl.e]\
  -w SP_MULTS_TEMP -create_update
analyze -f vhdl [list sp_mult.vhdl.e] -w SP_MULTS_TEMP -create_update
```

```
analyze -f vhdl [list sp_mult_csa.vhdl.e sp_mult_wallace.vhdl.e] \
  -w SP_MULTS_TEMP -create_update
```

Remember to remove the intermediate files:

```
rm -r ~/analyze_temp
```

7.2.5 Step 5: Create dc_shell-t script

The `dc_shell-t` script is used to create a design library for your DesignWare Building Block IP component categories from encrypted HDL files. It is recommended that you create a `dc_shell-t` installation script for each family of products. In the SuperParts case there are an `sp_adders.script` and an `sp_mults.script` in the `SP_ADDERS` and the `SP_MULTS` directories, respectively.

The `dc_shell-t` script contains a series of `analyze` commands. All HDL files that are part of the DesignWare Building Block IP product category must be analyzed for each new release of the Synopsys tools in order to create intermediate files that work with the particular version of the tool.

It is expected that your customers will run the provided script each time they receive a new version of Synopsys software.

For VHDL, the files include the VHDL packages, the entity declarations, and the architectures. The argument (file name) to the `analyze` command must reflect the inter-dependency between the various files; you must first analyze the packages (if any), then the entity declaration, and finally the architectures.

If an entity or architecture in one component family depends on a package in another component family, the family containing the package must be processed first.

The `-update` option permits users who have only Verilog HDL licences to analyze VHDL source files.

The `sp_adders.script` file might look like this:

```
analyze -f vhdl [list sp_add.vhdl.e] -w SP_ADDERS -update
analyze -f vhdl [list sp_add_fast.vhdl.e sp_add_small.vhdl.e] -w SP_ADDERS\        -
update
analyze -f vhdl [list sp_add_turbo.vhdl.e] -w SP_ADDERS -update
exit
```

Note that the file containing the entity declaration is analyzed before those containing the architectures. Also note that `SP_ADDERS` is a logical name of the design library.

Similarly, the `sp_mults.script` file might look like this:

```
analyze -f vhdl [list sp_mult_package.vhdl.e] -w SP_MULTS -update
analyze -f vhdl [list sp_mult.vhdl.e] -w SP_MULTS -update
analyze -f vhdl [list sp_mult_csa.vhdl.e  sp_mult_wallace.vhdl.e] \
  -w SP_MULTS -update
exit
```

Note that the package `sp_mult_package.vhdl.e` is analyzed first. The `sp_mult.vhdl.e` file, which contains the entity declaration, is analyzed next. Finally, the VHDL files containing the architectures are analyzed. As with `SP_ADDERS`, `SP_MULTS` is a logical name of the design library.

7.2.6 Step 6: Create installation instructions and script

The activities involved in this step are left to your discretion. It is important that the installation instructions and scripts follow the guidelines in the next section, Installing DesignWare Building Block IP Component Libraries.

It is recommended that any installation scripts be called `install` and that installation instructions be put in a README file. For example, our case study vendor, SuperParts, Inc., created `install` scripts and README files in the `src` subdirectory and in the subdirectory corresponding to each of its families. See Figure 7-3 on page 78.

7.3 Installing DesignWare Building Block IP Component Libraries

As a DesignWare Building Block IP component developer, you are responsible for providing detailed instructions to the user for proper installation of your components. This section recommends some basic installation steps to ensure the proper installation of Foundation components. You may consider providing your customers with an `install` script to automate all loading and installation tasks.

The installation procedure includes the following tasks:

1. "Step 1: Load the installation tape" on page 82
2. "Step 2: Verify that the tape was loaded successfully" on page 82
3. "Step 3: Create design libraries for each family" on page 82
4. "Step 4: Instruct users to edit setup file (Important!)" on page 83

These tasks are detailed in the following section. We continue with our SuperParts example to illustrate the installation procedure. It is assumed that the root directory for DesignWare Building Block IP component libraries is `/dw`.

7.3.1 Step 1: Load the installation tape

Provide the user with instructions for loading the contents of the tape or other distribution medium. The contents of your installation tape must be loaded in the DesignWare root directory.

7.3.2 Step 2: Verify that the tape was loaded successfully

Instruct the user to verify that the tape was loaded successfully; for example:

```
% ls /dw/SuperParts/V1.0/src
SP_ADDERS SP_MULTS install README

% ls /dw/SuperParts/V1.0/lib
sp_adders.sldb sp_mults.sldb
```

7.3.3 Step 3: Create design libraries for each family

Instruct the user to create a design library for each family. The following tasks must be performed for each family:

1. Create a directory for the family under the lib subdirectory. For example, for the `SP_ADDERS` family enter

```
% mkdir /dw/SuperParts/V1.0/lib/SP_ADDERS
```

2. Edit the system setup file located at

   ```
   synopsys_root/admin/setup/.synopsys_sim.setup
   ```

   to include the path name of the new design libraries as follows:

   ```
   SP_ADDERS: /dw/SuperParts/V1.0/lib/sp_adders
   SP_MULTS: /dw/SuperParts/V1.0/lib/sp_mults
   ```

3. For each product family, change the directory to the source directory containing the HDL description of the design libraries and execute the `dc_shell-t` installation script file.

   For example, for the two product families in the SuperParts example:

   ```
   % cd /dw/SuperParts/V1.0/src/SP_ADDERS
   % dc_shell-t -f sp_adders.script
   % cd /dw/SuperParts/V1.0/src/SP_MULTS
   % dc_shell-t -f sp_mults.script
   ```

4. (Optional—for Synopsys compiled simulation users only) Create object files for each family by using the `gvan` command with the `-c` option. Mimic the order of compilation in the corresponding installation script. For example:

   ```
   % gvan -c -w SP_ADDERS sp_add.vhdl.e
   % gvan -c -w SP_ADDERS sp_add_fast.vhdl.e
   % gvan -c -w SP_ADDERS sp_add_turbo.vhdl.e
   ```

The order of compilation is the same as in `sp_adders.script`.

Object files for compiled simulation are platform-dependent. If your network includes a variety of platforms, you must log in to each platform and repeat Step 4.

7.3.4 Step 4: Instruct users to edit setup file (Important!)

Instruct the user to edit the Synopsys set up file to include the path name of your DesignWare Building Block IP components to the `search_path`, and to modify `link_library` and `synthetic_library` variables accordingly. For example, a user who wishes to use the adders and multipliers from SuperParts must modify these variables as follows:

1. Add the directory containing the synthetic library database file to the `dc_shell-t` variable `search_path`. For `SP_ADDERS` do the following:

   ```
   set search_path [concat  $search_path \
     [list /dw/SuperParts/V1.0/lib]]
   ```

2. Add the synthetic library database file to the `dc_shell-t` variable `link_library`.

   ```
   set link_library [concat  $link_library [list sp_adders.sldb]]
   ```

3. Add the synthetic library database file to the `dc_shell-t` variable `synthetic_library`.

   ```
   set synthetic_library [concat  $synthetic_library \
     [list sp_adders.sldb]]
   ```

Refer to the *DesignWare User Guide* for more information on these variables.

A
Standard Synthetic Operators

Table A-1 lists the HDL operators that are mapped to synthetic operators in the Synopsys standard synthetic library `standard.sldb`. For information about the synthetic operators—input and output pins, associated modules, and so on—issue the following `dc_shell-t` command:

```
report_synlib standard.sldb
```

Table A-1 HDL Operators Mapped to Standard Synthetic Operators

HDL Operator	Synthetic Operator(s)
+	ADD_UNS_OP, ADD_UNS_CI_OP, ADD_TC_OP, ADD_TC_CI_OP
-	SUB_UNS_OP, SUB_UNS_CI_OP, SUB_TC_OP, SUB_TC_CI_OP
*	MULT_UNS_OP, MULT_TC_OP
<	LT_UNS_OP, LT_TC_OP
>	GT_UNS_OP, GT_TC_OP
<=	LEQ_UNS_OP, LEQ_TC_OP
>=	GEQ_UNS_OP, GEQ_TC_OP
if, case	SELECT_OP
division (/)	DIV_UNS_OP, MOD_UNS_OP, REM_UNS_OP, DIVREM_UNS_OP, DIVMOD_UNS_OP DIV_TC_OP, MOD_TC_OP, REM_TC_OP, DIVREM_TC_OP, DIVMOD_TC_OP
=, not=	EQ_UNS_OP, NE_UNS_OP, EQ_TC_OP, NE_TC_OP
<<, >> (logic) <<<, >>> (arith)	ASH_UNS_UNS_OP, ASH_UNS_TC_OP, ASH_TC_UNS_OP, ASH_TC_TC_OP ASHR_UNS_UNS_OP, ASHR_UNS_TC_OP, ASHR_TC_UNS_OP, ASHR_TC_TC_OP
Barrel Shift ror, rol	BSH_UNS_OP, BSH_TC_OP, BSHL_TC_OP BSHR_UNS_OP, BSHR_TC_OP
Shift and Add srl, sll, sra, sla	SLA_UNS_OP, SLA_TC_OP SRA_UNS_OP, SRA_TC_OP

Index

P

R

S

Z